# Scaling the Issue Window with Look-Ahead Latency Prediction

Yongxiang Liu[+], Anahita Shayesteh[+], Gokhan Memik[*] and Glenn Reinman[+]
+Dept. of Computer Science, University of California, Los Angeles, CA 90095-1596
*Dept. of ECE, Northwestern University, Evanston, IL 60208-3118
{yx,anahita,reinman}@cs.ucla.edu, memik@ece.northwestern.edu

## ABSTRACT

In contemporary out-of-order superscalar design, high IPC is mainly achieved by exposing high instruction level parallelism (ILP). Scaling issue window size can certainly provide more ILP; however, future processor scaling demands threaten to limit the size of the issue window.

In this study, we propose a dynamic instruction sorting mechanism that provides more ILP without increasing the size of the issue window. In our approach, early in the pipeline, we predict how long an instruction needs to wait before it can be issued, i.e. the waiting time for its operands to be produced. Using this knowledge, the instructions are placed into a sorting structure, which allows instructions with shorter waiting times enter the issue window ahead of those instructions with longer waiting times, preventing long-waiting instructions from clogging the issue queue.

The accuracy in predicting instruction waiting times directly determines the effectiveness of our sorting mechanism. While most instructions have deterministic execution latencies, predicting load execution times is more difficult due to cache misses and in-flight loads. Loads are particularly challenging since their execution time can vary significantly. In this study, we examine techniques to predict load execution time accurately, based on data reference history.

## Categories and Subject Descriptors

C. - Computer Systems Organization, C.1 - Processor Architectures, C.1.0 -General

## General Terms

Design, Performance

## Keywords

LHT, CLP, MNM, SILO, Instruction Sorting

## 1. INTRODUCTION

In contemporary out-of-order superscalar design, instruction-level parallelism (ILP) is mainly achieved in hardware by selecting as many independent instructions as possible for concurrent execution from the issue window. The size of the issue window places a bound on the amount of ILP that can be extracted from a dynamically scheduled, out-of-order processor.


*ICS'04*, June 26–July 1, 2004, Saint-Malo, France.


Scaling the issue window size can certainly provide more opportunity to discover independent instructions. However, the size of the issue window does not scale as architects plan towards future technology sizes, due to the complexity of wakeup and select logic [15]. Recent trends in microprocessor design have favored deeper pipelines and shorter cycle times [5]. This trend can impact the amount of logic architects are able to place on the critical path of the processor. The issue window is a critical structure, and therefore its size is likely to be impacted.

In a traditional processor, instructions enter the issue window sequentially. An instruction will enter the window even if it takes a long time to resolve its dependencies on input operands. Such instructions will likely clog the issue window and prevent instructions that are ready to issue from entering the window, which limits the ILP that the processor can achieve.

One recent study suggests dynamically removing instructions waiting on a long latency instruction to a separate buffer called Waiting Instruction Buffer (WIB) [8]. Another recent study proposed Cyclone[4], an effective broadcast free dynamic scheduler. Cyclone makes use of a switchback queue to delay the issue of instructions based on their deterministic latency, where loads are always predicted to hit in the level one cache. Both of these approaches handle load misses by consuming issue bandwidth to either send dependents of load misses to auxiliary storage (WIB) or to the head of the switchback queue (Cyclone). These techniques dynamically adapt to non-deterministic instruction latency, potentially consuming available issue bandwidth to shuffle dependent instructions. The scheduler still does not have any knowledge of when the load will actually complete.

In this paper, we propose a mechanism to predict nondeterministic instruction latency early in the pipeline. We explore this approach through a novel architecture that dynamically sorts instructions prior to the issue stage. Our approach first accurately predicts the waiting time that an instruction will endure before its operands are ready for execution. Instructions then enter the sorting structure, which consists of a number of differently sized FIFO queues. Instructions with longer waiting times (i.e. "slow" instructions) enter a FIFO queue of longer length, which will delay the instruction from entering the issue queue. Instructions with shorter waiting times (i.e. "fast" instructions) enter a FIFO queue with a shorter length, and are delivered to the issue queue with less, or no delay. All instructions are placed into a final FIFO queue, the Pre-issue Buffer, which then feeds the issue window in-order. Instructions with different latencies can enter the Pre-issue Buffer out-of-order with respect to one another. This effectively prevents "slow" instructions from clogging the issue window when there is available ILP in the application, without consuming available issue bandwidth.

Our latency predictor plays an important role in guiding the sorting process. While most instructions have deterministic execution latencies, predicting load execution latencies is more difficult. The latency of a load operation can vary significantly, ranging from the latency of the level 1 data cache to latencies beyond main memory that are possible in the face of bus contention. Though difficult, load latency prediction is the key component for our sorting mechanism to work efficiently and maximize ILP. This is because a large fraction of the executed instructions directly or indirectly depend on load operations, particularly in applications with frequent register spills and/or large data structures. As show in Figure 1, our simulations detect a substantial fraction of dependent instructions in the issue window when a cache miss occurs. These dependent instructions cannot be issued and thus, clog the issue queue while the load miss is satisfied.

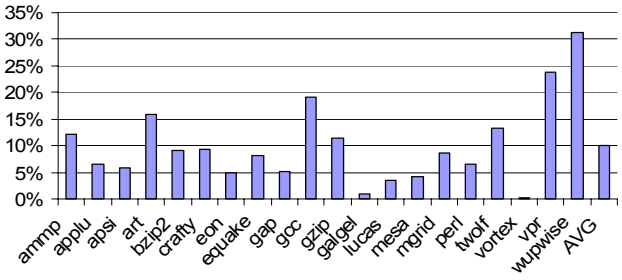


**Figure 1: Fraction of instructions currently in the issue window that are consumers of an outstanding load miss.**

Predicting load access times far in advance of the execution stream is not trivial. Modern processors typically employ a multilevel cache hierarchy. The time it takes to access different levels of the hierarchy varies significantly according to implementation. In addition to cache structure, whether a load hits or misses in the TLB during address translation can also impact load latencies. Even if we can have methods to determine if a load block is missing from the caches, its access time is still difficult to determine. Cache blocks can be in-flight (i.e. in the process of arriving from some level of the memory hierarchy) and therefore load latency can range from the L1 cache hit latency to a full memory access. Loads frequently alias with outstanding load misses, and their expected access time will be dependent on the in-flight load completion time. The latency of the in-flight load may be affected by possible structural hazards or contention in the pipeline or memory busses. In addition, stores may directly forward values to loads if the memory operations alias one another. In this paper we looked at several techniques to provide accurate load latency prediction.

We make the following contributions:

- We present techniques to efficiently predict the "waiting time" of all instructions, even those in a load dependency chain, early in the pipeline. We further demonstrate techniques to arbitrate between latency predictors.
- We test our latency prediction architecture by developing a simple sorting structure with a number of FIFO queues to enable instructions to enter the issue queue out-of-order. The queues delay instructions with longer waiting times and allow instructions with shorter waiting times to pass through quickly. The pre-issue buffer of this sorting structure provides a scalable approach to decoupling resource allocation and register renaming from instruction issue.
- We demonstrate the improvement achievable through instruction sorting that is assisted by latency prediction compared to increasing the issue queue size. By keeping the issue queue size smaller, and by performing latency prediction off the critical path, our approach provides a more scalable improvement in ILP.
- We show our load latency prediction can assist selectively replaying schemes, such as Cyclone, to predict load misses in advance, thus reducing the number of required replays.

The remainder of this paper is organized as follows. In Section 2 we describe the overall design of our architecture and the experimental methodology for evaluating our design. We present the design components on latency predictions in Section 3, and the instruction sorting engine in Section 4. The results from our simulations are presented in Section 5. We summarize related work in Section 6, which is then followed by the conclusions and highlights of future work.

# 2. OVERALL ARCHITECTURE AND EXPERIMENT METHODOLOGY

Figure 2 illustrates the overall architecture of our design. It consists of three major components: a Latency Prediction component, which estimates the waiting time of instructions, a sorting structure, which consists of a few FIFO sorting queues, and a Pre-issue Buffer (PB), where instructions are buffered before entering the issue queue. The PB provides a temporary storage space for the sorted instructions, but does not make use of wakeup and select logic, as in the case of the less scalable issue queue. Instructions are sequentially fed from the PB into the issue queue every cycle when space is available.

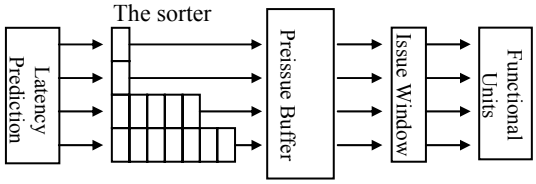


**Figure 2: the Overall Scheduling Architecture**

We make use of the SimpleScalar 3.0 tool set [1][2] to evaluate our design. We simulate 10 floating-point and 10 integer benchmarks that were randomly selected from the SPEC 2000 benchmarking suite. The applications were compiled with full optimisation on a DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital Unix V4.0. We simulate 100 Million instructions after fast-forwarding an application-specific number of instructions according to Sherwood et al. [17].

## 2.1 BASELINE

Table 1 shows the applications we used, including the percent of load instructions, the L1 and L2 data cache miss rates, and branch prediction accuracy. The L1 miss rate ("L1 MR") does not include in-flight data.

The IPC shown in Table 1 is from the simulation of baseline architecture which models a future generation microprocessor. We simulated our baseline using a 256-entry ROB and a 128-entry LSQ. The issue window is varied from 12 to 32 (12, 16, 24 and

32 entry). We model an 8-way out-of-order superscalar processor. The cache parameters are based on the P4 with an 8 KB, 32-Byte block size, 4-way set associative L1 data cache with 2 cycle latency. The L1 instruction cache is a 16KB, 2-way set associative cache with a 32-byte block size. The unified L2 cache is a 512KB, 64-Byte block size, 4-way set associative cache with a 12-cycle hit latency. 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. The latencies are: ALU 1 cycle, Integer MULT 5 cycles, Integer DIV 25 cycles, FP ADD/CMP/CVT 2 cycles, FP MULT 10 cycles, FP DIV 30 cycles, and SQRT 60 cycles. All functional units, except DIV and SQRT units, are fully pipelined allowing a new instruction to initiate execution each cycle. We use a 4K-entry combining predictor to select from a 4K-entry bimodal predictor and an 8K-entry gshare predictor. The minimum branch misprediction penalty is 20 cycles.

**Table 1: The Statistics of baseline benchmarks**

| Data set | %load | %L1 MR | %L2 MR | Branch Acc. | IPC(Win 32) | IPC(Win. 12) |
|---|---|---|---|---|---|---|
| ammp | 27.5 | 8.2 | 24.5 | 0.938 | 0.935 | 0.591 |
| applu | 30.2 | 18.9 | 16.4 | 0.917 | 1.123 | 0.728 |
| apsi | 24.4 | 2.6 | 1.9 | 0.907 | 1.431 | 1.098 |
| art | 26.9 | 40.6 | 66.3 | 0.927 | 0.714 | 0.324 |
| bzip2 | 28.4 | 0.8 | 9.2 | 0.992 | 2.683 | 1.421 |
| crafty | 30.2 | 5.5 | 0.3 | 0.919 | 1.020 | 0.902 |
| eon | 33.3 | 2.6 | 0.0 | 0.918 | 1.252 | 0.989 |
| equake | 40.7 | 18.4 | 31.3 | 0.948 | 0.420 | 0.280 |
| gap | 26.0 | 1.1 | 6.1 | 0.992 | 1.630 | 1.190 |
| gcc | 24.3 | 12.6 | 6.0 | 0.951 | 0.741 | 0.638 |
| gzip | 20.0 | 9.7 | 0.5 | 0.921 | 1.645 | 1.106 |
| galgel | 39.6 | 15.3 | 9.7 | 0.952 | 1.997 | 1.083 |
| lucas | 12.5 | 20.3 | 33.3 | 0.993 | 0.717 | 0.606 |
| mesa | 27.1 | 1.9 | 7.3 | 0.957 | 2.055 | 1.252 |
| mgrid | 31.7 | 15.4 | 17.7 | 0.957 | 1.753 | 1.382 |
| perl | 28.6 | 0.4 | 5.7 | 0.990 | 2.427 | 1.398 |
| twolf | 26.8 | 7.5 | 14.6 | 0.900 | 0.708 | 0.565 |
| vortex | 29.8 | 0.6 | 2.9 | 0.990 | 1.396 | 1.173 |
| vpr | 42.6 | 5.0 | 41.0 | 0.879 | 0.710 | 0.481 |
| wupwise | 22.6 | 3.2 | 31.6 | 0.960 | 1.861 | 1.074 |

# 3. LATENCY PREDICTION

Throughout this paper we use the following terms: *instruction waiting time* and *execution latency*. The *instruction waiting time* refers to the number of cycles that must elapse before an instruction's operands are ready for execution. The *execution latency* of an instruction refers to the time it takes for the functional unit to execute the instruction. It is referred to as *load access time* in the case of load operations.

## 3.1 Deterministic Instruction Waiting Time

Figure 3 illustrates the architecture we use to compute the waiting time of deterministic instructions. Similar to the instruction pre-scheduler in Cyclone[4], our design incorporates a timing table indexed by logical register names, which stores the time when logical register value is going to be produced. It allows four accesses in a row. The computation involves two steps:

Step 1, MAX Computation: Each instruction obtains its ready time by accessing a timing table indexed with its input operands, and take the maximum completion times.

Step 2, Waiting Time & Completion Time Computation: result from Step.1 is added to the instruction execution to get the ready time of the instruction's destination register. This result is then written back to the timing table. Completion time is calculated by comparing the ready time with a global time counter.

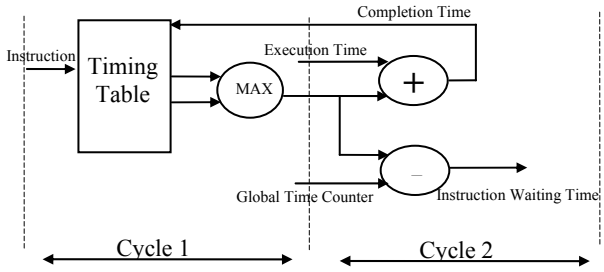


**Figure 3:Computation of Deterministic Waiting Time**

It is possible to have dependencies among instructions that are renamed in the same cycle. As in [4] we limit the dependency chain that can be handled by the timing table to two cascaded MAX computations. If a dependency-chain is longer than two, we force the computation to the next cycle, effectively stalling the front-end. Our simulations (in agreement with [4]) revealed that such stalls cause very little performance degradation, because longer dependency chains in the same cycle are not common in the SPEC2000 suite.

## 3.2 Load Access Time Prediction

Our load latency prediction architecture is made up of two components: a Latency History Table (LHT), which predicts latency using a last value predictor, and Cache Latency Propagation (CLP), which makes use of existing cache miss detection techniques [11]. The LHT can be accessed as early as instruction fetch as it only requires the instruction PC to make a prediction. The CLP can also be accessed early on in the pipeline, but requires address prediction (indexed via PC).

### *3.2.1. Latency History Table (LHT)*

Value prediction [10] has been used in the past to predict load values. We use the same design to predict load latencies. The LHT predicts whether or not a given load will experience the same latency as the last access of that load. As shown in Figure 7, the LHT has 2K entries, is indexed by PC, and has 3-bit confidence counters to avoid less confident predictions. The confidence threshold to predict is four.

If the LHT cannot make a confident prediction, we guess the L1 data cache latency for the load latency. Note that this is a conservative assumption, but our intention is not to degrade performance. If the waiting time of an instruction is predicted to be longer than the actual delay, the instruction will be unnecessarily delayed at the sorting stage, which is likely to degrade performance if this instruction is on the critical path of the application. On the other hand, if we underestimate the waiting time, we may miss the opportunity to reduce issue queue clogging, but we will not degrade performance below that of our baseline.

LHT can accurately predict loads that frequently hit in the L1 cache. However as shown later, it is only able to predict 24% of misses in both L1 and L2 data caches. Our arbitration policy ensures that the LHT outperforms a scheme that always predicts a cache hit, by capturing L1 misses with predictable latency. The LHT is not good at handling aliases to in-flight loads.

### *3.2.2 Cache Latency Propagation (CLP)*

A load hit can be handled like a deterministic instruction when calculating the *instruction waiting time*. The execution latency of a load that will hit in the cache can simply be set to the L1 cache hit latency. However, a load miss needs special handling because it impacts not only its own dependent instructions, but also any subsequent aliasing loads and their dependent instructions. For instance, if load A misses, and subsequent load B aliases with A, A may hide some of the latency seen by B. This will, in turn, impact B's dependent instructions.

Our Cache Latency Propagation (CLP) scheme identifies cache misses, and then propagates the completion time of cache misses to any aliasing loads via a structure that stores the Status of In-flight Loads (SILO). The CLP approach needs an address predictor, which produces load addresses that are used by a cache miss predictor.

***Address Prediction/Precomputation***

Our address predictor makes use of a hybrid address predictor similar to that proposed by Wang and Franklin [19]. It consists of a 2K entry stride predictor and a 4K entry Markov predictor. Prediction is guided by 3-bit confidence counters to reduce mispredicted addresses. The confidence threshold is four.

For many loads, the address may be precomputed if the input operand to the load is ready far enough in advance of the execution of the load operation. However, a precomputation approach needs to monitor the status of load address operands, and requires additional register file ports or a shadow register file to access load address operand values. We do not implement this approach in our design, but recognize that more sophisticated techniques are available for look-ahead address generation. In later sections, in addition to performance gain using stride/Markov predictors, we show the potential extra gain if an accurate look-ahead address generator is used.

***Cache Miss Predictor***

In CLP, we look for a more rigid cache miss predictor as we have additional knowledge, i.e. load addresses that are produced by the address predictor.

Memik et al. [11] used cache miss prediction to reduce the cache access times and power consumption in a processor with multiple cache levels. However, in their approach, the detection engine was accessed along with the data cache access, which is too late for our approach. We use a similar technique for cache miss detection using a small hardware structure to quickly determine whether or not an access will miss in the cache. It stores information about the block address placed and removed from the caches. We use the hybrid cache miss detector as described in [11]. We assume a two level cache structure in our architecture, and therefore make use of three detection engines to indicate whether a load access is a miss in the L1 cache, a miss in L2 cache, or a miss in the TLB. The miss detection engine never indicates a miss when the block is in the cache, but it can wrongly indicate a hit when the block is actually not in the cache (i.e. there are almost no false predictions on the detected misses)[11]. This matches our desire to be conservative in latency estimation, as we will not guess a longer latency if the load is currently in the cache.

***Status of In-flight Load (SILO) Structure***

The miss detection engine cannot detect in-flight data accesses. For instance, when the engine claims a block is a miss in the cache, two cases are possible:

a) There is an earlier load operation on the same block that has been renamed but has not reached the cache access stage of the pipeline.

b) The access has already started but the data block has not yet arrived in the cache.

Figure 8 indicates the frequency and importance of capturing in-flight loads. We design the Status of In-flight LOads (**SILO**) structure to capture the initial miss and propagate its completion time to the aliasing loads. Without the SILO, a load instruction that aliases with another *in-flight* load instruction may be mistaken as a miss.

SILO is a small, cache-like structure addressed by the predicted address of load instructions. Our SILO is similar to the PAT structure presented in [12]. PAT stores the time remaining for in-flight data blocks to reach the cache. In SILO, on a load miss, we store its completion time and propagate the time to later aliasing loads, if any exists.

Speculative loads that have been renamed but are squashed before execution may pollute the SILO. If such loads record their completion times into the SILO, a subsequent aliasing load will wrongly conclude that an earlier load has accessed the same cache block. We solve this problem by maintaining two versions of the SILO: One is updated at issue time like a PAT [12], and the other is updated speculatively at prediction time. On a branch misprediction, we use the nonspeculative SILO to recover the speculative SILO. We use the issue time SILO to invalidate erroneous entries.

Often, the L2 cache block size is larger than the L1 cache block size to take better advantage of locality and reduce main memory accesses. For instance, in P4 [5], the L2 cache block size is 64B, twice the L1 block size. On a load miss, not only the L1 block but its adjacent block is brought into the L2 cache. We maintain two separate SILO structures (and two separate recovery structures) to addresses this. One SILO tracks the L1 blocks, and the other tracks their adjacent L1 block. In the latter SILO, the value stored is the time for the block to arrive in the L2 cache. If both SILOs have a valid hit, we defer to the L1 cache's SILO.

Our simulation results indicate that there is little need to track the in-flight status of TLB misses.

We use 8 entries for each SILO structure. The SILO tag length is 25 bits. Each record is 10 bits. Hence, each SILO structure we use is not more than 35 Bytes.

***Overall Algorithm of CLP***

The SILO is probed using the predicted address, If a load hits in the SILO, it skips Step One, and directly performs Step Two by recording the completion time obtained from the SILO into the timing table. In this way, the completion time is propagated to the dependents of the aliasing loads.

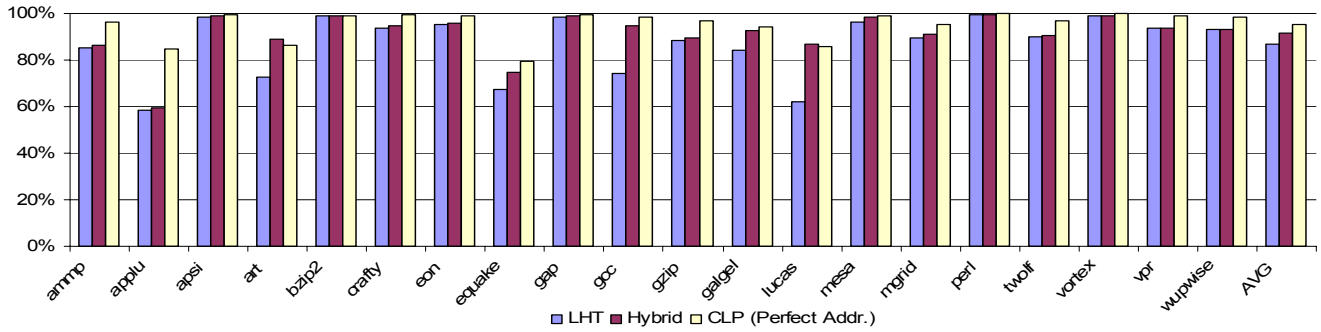


**Figure 4: The Prediction Accuracy Rate of Load Access Times**

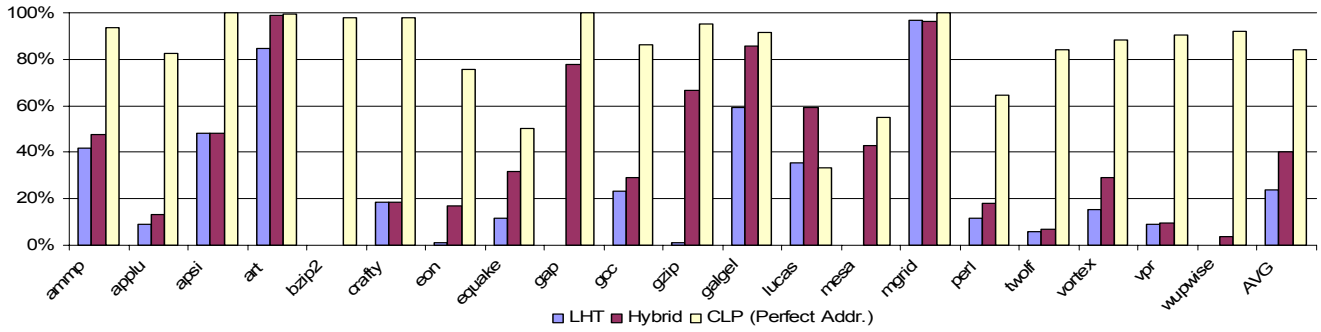


**Figure 5: The Percentage of Load Misses (includes misses either in the L1 or the L2) That Are Correctly Predicted**

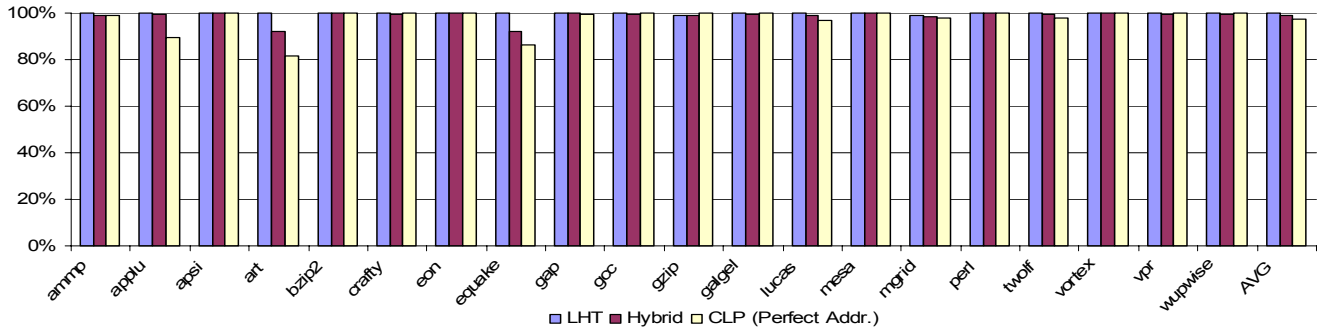


**Figure 6: The Percentage of L1 Hits That Are Correctly Predicted**

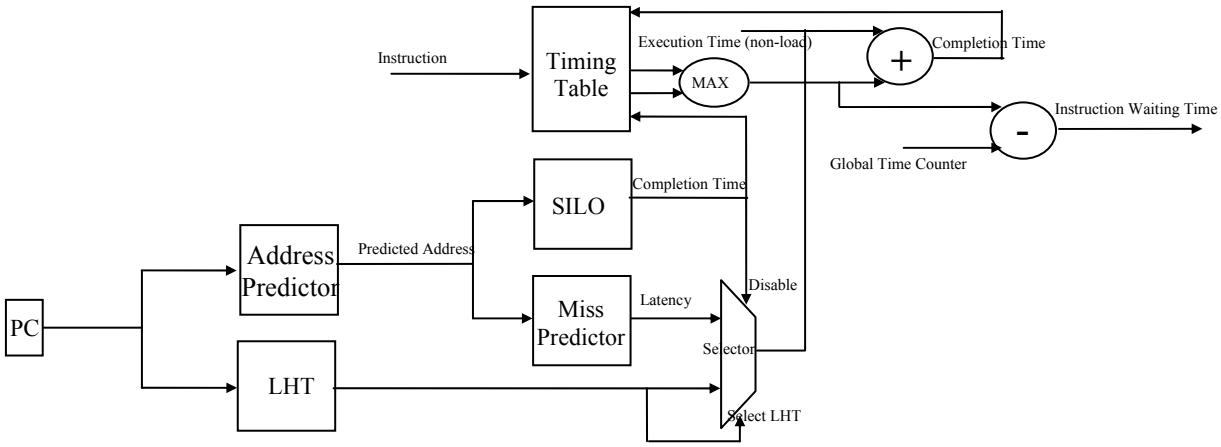


**Figure 7: Block Diagram of Waiting-time Predictor**

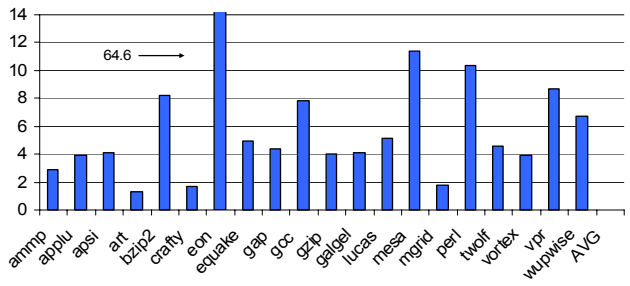


**Figure 8: The Average Number of Loads that Alias In-Flight Memory Accesses**

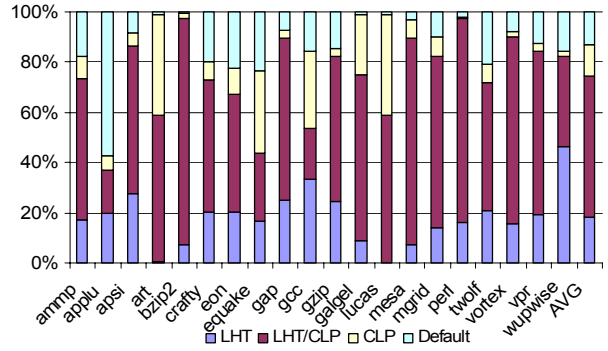


**Figure 9: The breakup by Prediction Methods in the Hybrid Approach**

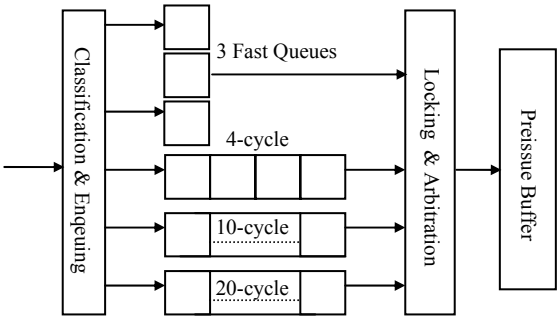


**Figure 10: A Sample Configuration of the Sorting Engine**

The cache miss predictor is accessed in parallel with SILO. SILO is given priority over the cache miss predictor, so that loads aliased with in-flight loads can always receive the propagations from SILO. The cache miss predictor produces a load access time by arbitrating among three levels of miss detection structures as shown in Table 2. Note that predictions from the different cache detector may not agree. For instance, the L1 structure claims that there may be a hit, while the L2 structure claims that there is a miss. Because the miss detection structure claims a miss ("yes") only when it is sure, but maybe ("no") when it is unsure, we use the prediction from the L2.

**Table 2: Prediction based on Output of Miss Detection Engine**

| Cases | Dl2 Miss? | Dl1 Miss? | Tlb Miss? | Prediction |
|---|---|---|---|---|
| Case A | Yes | Don't Care | Don't Care | Mem. Lat. |
| Case B | No | Don't Care | Yes | TLB M Lat. |
| Case C | No | Yes | No | L2 Lat |
| Case D | No | No | No | L1 Lat |

Assumption: Mem. Lat. > TLB Miss Lat > L2 Lat. > L1 Lat.

The CLP approach is efficient at capturing loads misses As shown in Figure 5, the CLP with an ideal address predictor is able to capture 84% of the memory accesses. The CLP also accurately captures the L1 cache hits. Figure 6 shows that CLP captures 97% of the L1 cache hits, which is slightly outperformed by the LHT. Overall, CLP can predict up to 95% of L1 cache hits.

However, CLP is limited by address prediction. Our simulations show that a realistic address predictor accurately covers 68% of addresses on average. We use CLP in a hybrid approach for those loads that are not predictable by LHT, but are address predictable.

### *3.2.3 Hybrid Approach*

In our hybrid design, LHT is given priority when LHT and CLP both predict confidently, as accessing the CLP structure is more expensive in terms of hardware cost. If both techniques fail, the default value of the L1 hit latency is used.

Figure 7 shows the overall architecture of the hybrid load latency predictor, which is combined with the structures for computing instruction waiting time. Predictor look-ups start in parallel with the rest of the pipeline as early as the fetch stage. This can hide the latency of the prediction structures, as the prediction starts in fetch and is required during rename. The predictors allow four accesses per cycle. Prior work has demonstrated multi-access techniques to predictors [9]. Address predictors are updated speculatively with predicted values when a prediction is made. In the commit stage, confidence counters are updated non-speculatively for the address predictor and LHT. We also update the last latency seen for the LHT and the stride of the address predictors in commit. Cache changes are updated to the cache miss predictors in the write-back stage.

Figure 9 shows the distribution of predictions by the three prediction methods, namely LHT, CLP and default (always guess the L1 latency). The loads that are both predictable by LHT and CLP are categorized as "LHT/CLP". The benchmarks that have less LHT coverage but are address-predictable can benefit from the hybrid approach. As shown in Figure 5, the Hybrid approach is able to capture 40% of the memory accesses and close to 99% of the L1 cache hits. Its overall prediction rate is 92%.

## 4. INSTRUCTION SORTING

Once its waiting time is predicted, an instruction is classified and placed into one of the FIFO queues in the sorting engine. The primary function of the FIFO queues is to hold the instructions until their waiting time has elapsed. Instructions with very short waiting times are placed into fast queues and visa versa. Figure 10 shows a simple configuration where there are 3 fast queues, which let instructions go immediately in the next cycle, and 3 slower queues, with hold instructions for 4, 10 and 20 cycles respectively. Instructions progress one slot per cycle, then are ultimately released into the Pre-issue Buffer (PB).

During classification, we round down waiting time to the closest latency class. For example, an instruction with waiting time of 12 is rounded down to the 10-slot queue. This ensures that instructions are not held for more than their waiting time. Hence, our sorting engine will not affect instructions on the critical path. Instructions are placed in a round-robin fashion in the same class. The front end of the processor is stalled if an instruction cannot find an available queue.

It is possible that an instruction reaches the queue head while its parent instruction is still inside FIFOs due to a waiting time misprediction. If we let the child instruction leave for the PB first, deadlock can occur. If the issue window is filled with dependents of instructions in the scheduling queue, forward progress will become impossible.

We address this problem using a small structure called Locking Table, which is indexed by physical registers. An instruction sets the "lock" flag for its destination physical register when it is enqueued, and resets the flag when it is dequeued. A subsequent instruction that consumes the register will check the locking table for a "lock" flag when it reaches the head of the FIFO. An instruction is released into PB if no "lock" is present. We limit the number of sorting queues to 8, to keep the accesses to the locking table less than 16 at each cycle (two accesses per instruction). As described above, our locking table is a very simple structure that returns a single bit for each register index, and can be easily replicated to reduce port costs.

**Table 3: Effectiveness of the Sorting Engine**

| Old Seq | Instruction | RT | EL | WT | New Seq |
|---|---|---|---|---|---|
| 1 | Ldt f25,-320(r9) | 0 | 12 | 1 | 1 |
| 2 | Addq r3,r24,r3 | 0 | 1 | 10 | 7 |
| 3 | Ldt f18,0(r3) | 0 | 12 | 12 | 8 |
| 4 | Mult f25,f18,f18 | 1 | 10 | 23 | 9 |
| 5 | Addt f0,f18,f0 | 1 | 2 | 35 | 10 |
| 6 | Stt f0, 0(r25) | 2 | 0 | 36 | 11 |
| 7 | Ldq r3,-32(r27) | 3 | 6 | 1 | 2 |
| 8 | Ldt f30,-256(r9) | 3 | 0 | 1 | 3 |
| 9 | Addq r3,r24,r3 | 3 | 1 | 7 | 5 |
| 10 | Ldt f16,0(r3) | 3 | 164 | 9 | 6 |
| 11 | Mult f30,f16,f16 | 4 | 10 | 172 | 12 |
| 12 | Addt f0,f16,f0 | 4 | 2 | 186 | 13 |
| 13 | Stt f0,0(r25) | 5 | 0 | 187 | 14 |
| 14 | Ldq r3,-24(r27) | 6 | 12 | 1 | 4 |

Symbols: RT(Renaming Time) -- relative time when instruction is renamed. EL (Execution Latency) -- instruction execution latency. WT(Waiting Time) -- instruction waiting time.
Dependency Chains: 2->3->4->5->6, 1->4->5->6, 7->9->10->11->12->13, 8->11->12->13, 14 is independent.

Table 3 shows the sequence of instructions before and after sorting from one of our experiments. The sorting queue configuration has 3 0-slot queues, 2 5-slot queues, 1 10-slot queue, 1 20-slot queue, and 1 150-slot queue. The 14 instructions above are renamed in 6 consecutive cycles as shown in the third column. There are four chains of instruction dependencies: a). 2,3,4,5 and 6. b). 1,4,5 and 6. c). 7,9,10,11,12 and 13. d) 8,11,12 and 13, e) 14. From the predicted values, we notice instructions 1, 7, 8 and 14 have their operands ready sooner than any other instructions. Our sorting engine successfully places these instructions ahead of others. Instruction 9 and 10 observe waiting times of 7 and 9 cycles respectively. As mentioned, their waiting times will both be rounded down to the nearest sorting class, in this case 5 cycles, and they are both placed into a 5-slot queue. Following the same analysis, instruction 2,3,4,5,6 are placed into a 10-slot queue. Hence, after sorting, instructions 9 and 10 are in front of 2,3,4,5 and 6. Instructions 11,12 and 13 observe large waiting times as a result of memory access. They are placed into the 150-slot queue, and hence, they come out last. Overall, the sorting engine effectively places the "slow" instructions after the "fast" instructions.

# 5. EXPERIMENTS AND RESULTS

## 5.1. Waiting Time Classification

We profiled all 26 SPEC2000 benchmarks to obtain the histograms of instruction waiting time. The histogram graphs are selectively presented in Figure 11. Each graph is zoomed in its right-upper corner to reflect the distributions in lower range latencies. The histograms indicate that a large fraction of instructions have waiting times of more than 150 cycles, therefore we simulated a slow queue of 150 slots to buffer these instructions. Similarly, we need a few fast queues to expedite the instructions with very short waiting time, and a few intermediate queues with 5, 10, and 20 slots. We need fewer sub-queues for the slow FIFOs, and more for the fast FIFOs, as there is more buffering capacity in a slow FIFO. We simulated three 0-slot queues, two 5-slot queues, one 10-slot queue, one 20-slot queue, and one 150-slot queue.

There are wide variations among the histograms of all benchmarks. We chose a simple sorting engine for our experiments here, but more sophisticated sorting engines will be explored as future work.

## 5.2. IPC Speedup over Baseline

We perform several experiments to measure the IPC speedup from our approach over a baseline with the same configuration. We use the parameters described in prior sections for the hardware components. All prediction structures are accessed off the critical path, during instruction fetch. We lengthen the pipeline by two additional cycles to account for enqueuing and dequeuing delays.

Figure 12 shows the speedup for a 12-entry issue queue over a baseline architecture with the same issue queue size, but without the extra branch misprediction penalty. The sorting engine guided by LHT shows an average speedup of 23%. The hybrid predictor consists of the LHT predictor and CLP with a realistic address predictor. With the hybrid approach, the sorting engine improves IPC by 28% over that of the baseline. CLP with an ideal address predictor shows a speedup of 34%, which shows potential improvement provided by CLP.

The best speedup is seen by art, which achieves 87%, 106%, and 107% speedup respectively for a LHT, Hybrid, and CLP (Perfect Addr) guided sorting engine. As shown in Table 1, art has a large amount of cache misses. This is also confirmed in Figure 11. The dependent instructions of these misses need to wait a substantial amount of time. In our approach, the cache misses are detected, and their dependents placed into the "slow queue", giving way to the "fast" instructions to the issue window. As the Hybrid predictor can more accurately detect misses, additional IPC speedup is observed. Further speedup is observed for CLP with ideal address predictor as it is able to predict cache misses more accurately.

Some benchmarks achieve less speedup. For example, gcc and crafty observe only 5-9% speedup. Table 1 shows that these two benchmarks have less cache misses than other benchmarks. Furthermore, Figure 11 confirms that gcc has only a few instructions that have a very long waiting time. While gcc has a large number of instructions that have shorter waiting times, our sorting engine is not able to differentiate these instructions due to our selection of queue sizes. As a consequence, the speedup from these benchmarks is small.

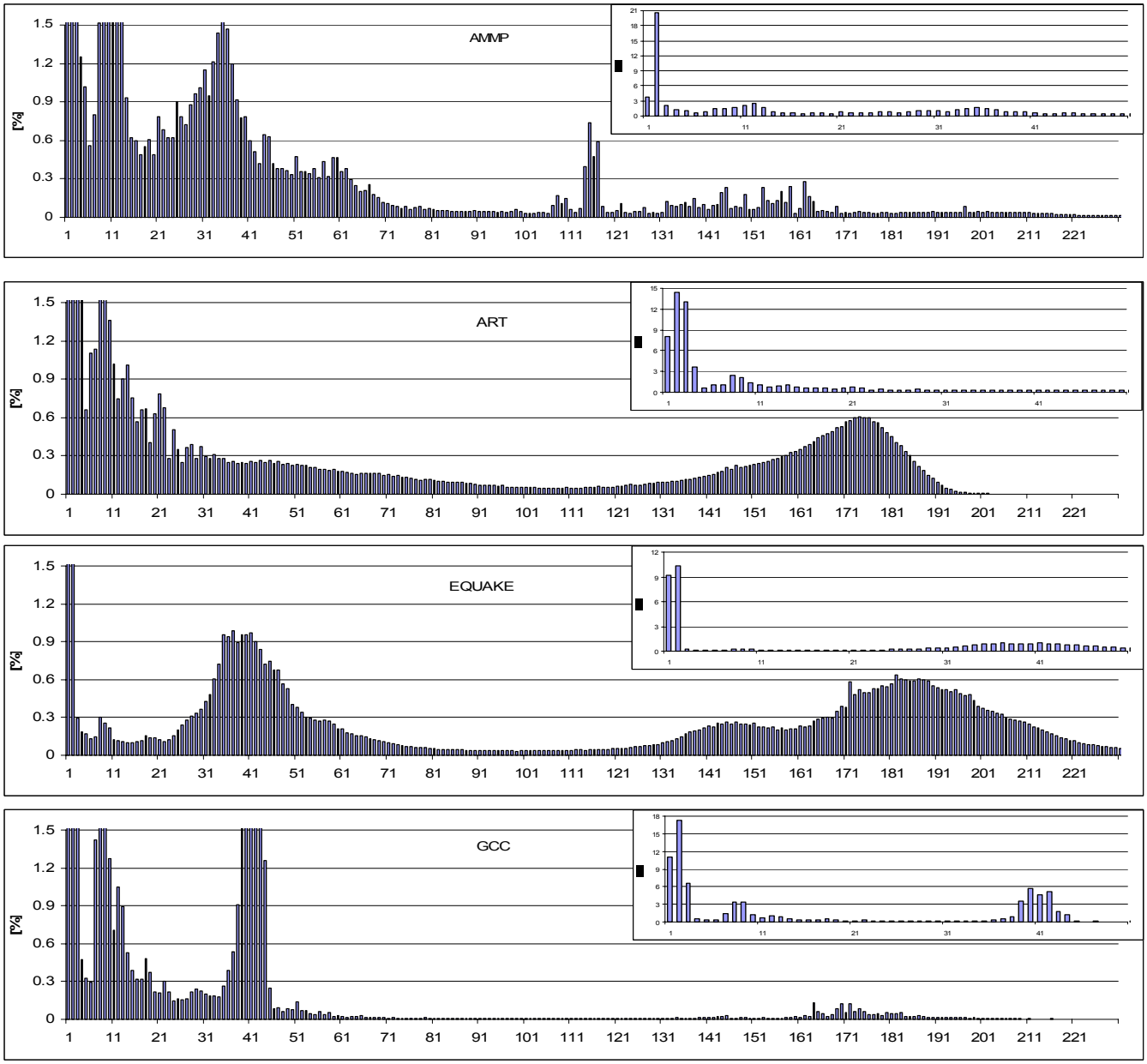


**Figure 11: The Histogram of Predicted Waiting Time for Selected Benchmarks**

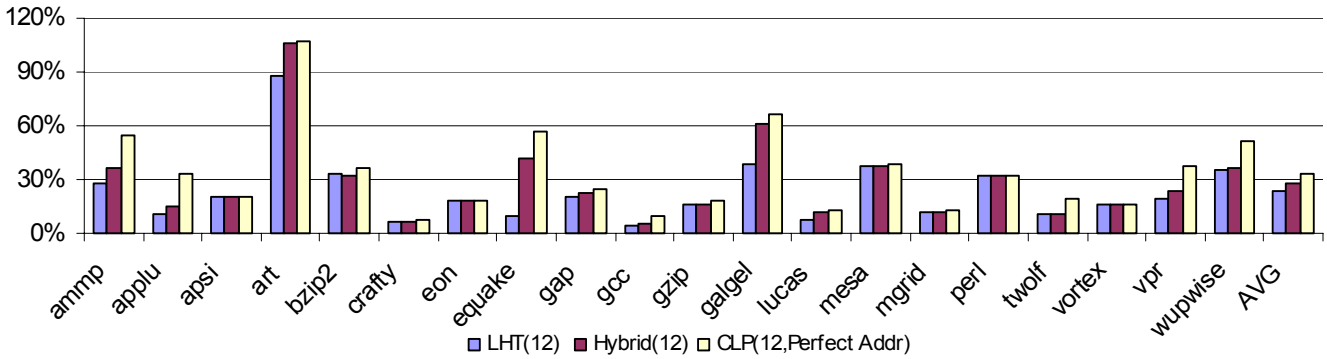


**Figure 12: The IPC Speedup with 12-entry Issue Queue over a Baseline Configuration**

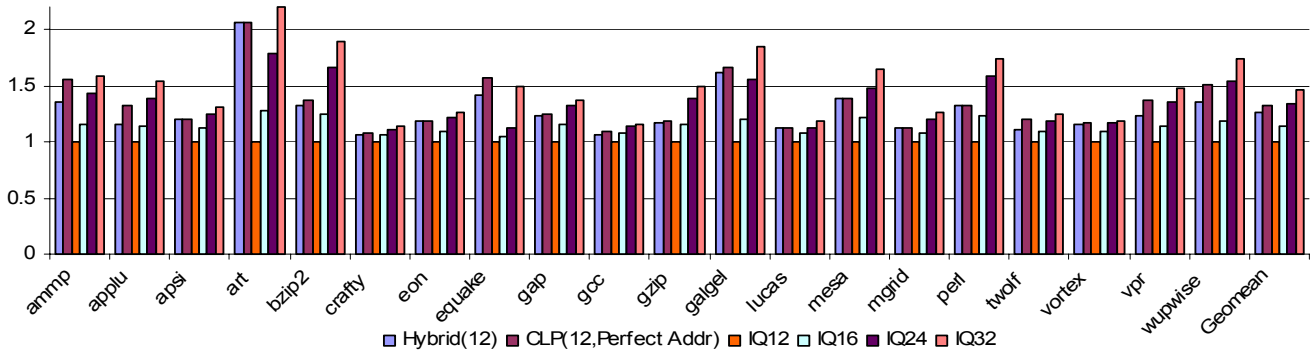


**Figure 13: Comparing the Normalized IPC of 12-entry Issue Window Using Prediction-guided Instruction Sorting with Baselines of 12, 16, 24, and 32 Issue Window**

Benchmarks that have higher misprediction rate tend to have less speedup. As shown in Figure 4 and Figure 5, lucas has a poor overall prediction rate, and a poor rate at identifying cache misses. Thus, lucas observes a speedup of only 10%.

Our approach provides sustained improvement to applications as the issue queue is scaled. For an issue queue of 32 entries, our approach provides an average 15% improvement over a baseline architecture with a 32-entry issue queue.

## 5.3. Issue Window Scaling Performance

Figure 13 shows the scaling effect by comparing with an issue window of 32, 24, 16 and 12 entries. The IPC values are normalized with respect to a baseline of 12-entry issue window. It is observed that our sorting engine guided by the Hybrid approach scales the performance of the 12-entry issue window within 95% of a 24-entry issue window in the baseline.

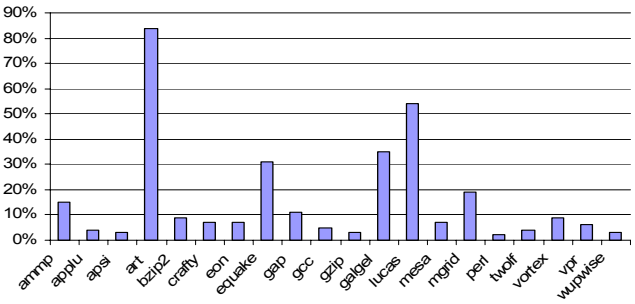


**Figure 14. Percentage of Replays Eliminated in Selective Replay Scheduling Using Load Latency Predictions**

## 5.4. Assisting Speculative Schedulers with Look-ahead Load Latency Prediction

Speculative schedulers, such as Cyclone, schedule load dependents ahead of time by assuming the load will hit in the data cache. If a load misses, its dependent instructions are selectively replayed [5,6] every N cycles until they receive the load data (where N is chosen to balance the number of replays with the added latency of doing replays). Our load prediction techniques can assist selective replay schemes to reduce the number of replays caused by cache misses by predicting load misses early in the pipeline. In this experiment, we compare a speculative scheduler scheme, which replays every 12 cycles (based on the L2 cache latency) with and without load latency prediction techniques. As shown in Figure 14, our results show an average 16% reduction in replays when latency prediction is used.

# 6. RELATED WORK

Several pre-scheduling techniques have appeared in the literature. Palacharla's dependence-base FIFOs [15] try to detect chains of dependent instructions and queue them into a set of FIFOs. In this way, only instructions at the FIFO heads are monitored for execution. Canal et al. [3] have proposed similar concept, with a different implementation. Michaud et al [13] have proposed to optimize instruction ordering based on data dependencies. The ILDP processor further refines dependence-based scheduling, by including instruction set support for describing dependent instruction sets [7]. LeBeck's WIB [8] scheduler moves long latency instructions into a waiting instruction buffer (WIB). Morancho et al uses a similar approach to move dependent operations following long latency instructions out of the instruction window[14]. Raasch et al [16] propose to break the instruction queue into segments forming a pipeline. The flow of instructions is governed by a combination of data dependencies, and predicted operation. Ideally instructions reach the final segment, where they can be issued when their operands are available.

Counterflow processors are proposed as an alternative to the superscalar approach by using highly localized communication to resolve the scheduling issues [18]. In the counterflow pipeline, instructions and data flow in opposite directions on circular queues. When instructions pass their input operands, they capture the data. Once an instruction has captured all its operands, it locates a functional unit, and leaves the queue to begin execution. Ernst et al's scheduler Cyclone [4] also uses decentralized dependence analyses to design a broadcast free scheduler. Cyclone predicts the waiting times of instructions, which are then placed into a countdown queue where they will be delivered to the execution engine.

Our approach uses a similar concept of timing instructions, to dynamically sort instructions prior to issue stage Similar to Cyclone we predict the waiting time of instructions far in advance at the stage of renaming. Unlike most previous studies, however, we look into load access time predictions. Our approach incorporates miss and in-flight load propagation structures, which can accurately predict load misses and propagate the times to their dependents.

The Alpha 21264 uses a global 4-bit history counter to determine whether a load hits in the cache[6]. However, it is difficult to accurately predict based on global history. The concept of load miss detection was proposed by Memik et. al. [11][12]. They use the cache miss detection engine (CMDE), and a previously accessed table (PAT) to determine the latency of memory accesses. This latency is used to issue instructions timely.

The accurate prediction on instruction waiting time enables our approach to be free of replaying on a missed load. Unlike previous research, such as WIB [8] and Cyclone [4], our approach does not waste issue bandwidth on replays. In addition, our load access time predictors can assist schemes like Cyclone to reduce excessive instruction replays.

# 7. CONCLUSIONS AND FUTURE WORK

In this study, an instruction sorting engine guided by proactive waiting time prediction is proposed. We have developed novel schemes to predict load access time accurately. Combined with waiting time computation, we are able to estimate the waiting time of instructions accurately. Guided by the estimated waiting time, our sorting engine can efficiently sort "fast" instructions ahead of "slow instructions". Simulation results show that our approach is able to exploit significantly more application ILP than a comparably sized issue window.

We consider several interesting follow-ups on this work. One direction is to improve load prediction accuracy, e.g. improving address prediction to better utilize the CLP's capability of predicting caches misses. It would also be interesting to exploit a better sorting mechanism or combine our prediction techniques with a pre-scheduler such as Cyclone.

# 8. REFERENCES


[1] T. Austin, E. Larson, D. Ernst. SimpleScalar: an Infrastructure for Computer System Modeling, IEEE Computer, Volume 35, Issue 2, Feb 2002.

[2] D. C. Burger and T. M. Austin, The Simplescalar Tool set, version 2.0, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[3] R. Canal and A. Gonzalez. A Low-Complexity Issue Logic. In Proceedings of the 2000 International Conference on Supercomputing (ICS 2001), May 2001.

[4] D. Ernst, A. Hamel, T. Austin, Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay. In Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-30), Jun. 2003.

[5] 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.

[6] R. E. Kessler. The Alpha 21264 Microprocessor. IEEE Micro, 19(2):24--36, March-April 1999.

[7] H. Kim and J. E. Smith. An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing. In Proceedings of the 29th International Symposium on Computer Architecture (ISCA-29), May 2002.

[8] 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 International Symposium on Computer Architecture (ISCA-29), May 2002.

[9] S. Lee and P. Yew, "On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors", International Conference on Parallel Architecture and Compiler Techniques (PACT2000).

[10] M.H. Lipasti, C.B. Wilkerson and J.P. Shen, Value Locality and Load Value Prediction. In Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-96), October 1996.

[11] G. Memik, G. Reinman, W. H. Mangione-Smith, Just Say No: Benefits of Early Cache Miss Determination, In Proceedings of the 9th IEEE/ACM International Symposium on High Performance Computer Architecture (HPCA-9), Feb. 2003.

[12] G. Memik, G. Reinman, and W. H. Mangione-Smith, Precise Scheduling with Early Cache Miss Detection, CARES Technical Report No. 2003_1.

[13] P. Michaud and A. Seznec, Data-flow prescheduling for large instruction windows in out-of-order processors. In Proceedings of the 7th International Symposium on High Performance Computer Architecture, (HPCA-6), Jan. 2001.R.F.

[14] E. Morancho, J. M. Llaberia and A. Olive. Recovery Mechanism for Latency Misprediction, In Proceedings of the 2001 International Symposium on Parallel Architectures and Compilation Techniques (PACT-2001), September 2001.

[15] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity effective Superscalar Processors, In Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-24), June 1997.

[16] S. Raasch, N. Binkert, and S. Reinhardt. A Scalable Instruction Queue Design Using Dependence Chain, In Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29), May 2002.

[17] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT 2001), Sep. 2001, Barcelona, Spain.

[18] R. F. Sproull, I. E. Sutherland and C.E. Molnar, "The Counterflow Pipeline Processor Architecture" IEEE Design and Test of Computers, pp. 48-59, Vol.11, No.3, Fall 1994.

[19] K. Wang and M. Franklin, Highly accurate data value prediction using hybrid predictors. In Proceedings of the 30th Annual International Symposium on Microarchitecture (Micro-30), Dec. 1997.