Appears in the 30th Annual International Symposium on Computer Architecture (ISCA-2003), June 2003

Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay

Dan Ernst, Andrew Hamel, and Todd Austin
Advanced Computer Architecture Lab
University of Michigan
Ann Arbor, MI 48109
{ernstd, ahamel, austin}@eecs.umich.edu

Abstract

To achieve high instruction throughput, instruction schedulers must be capable of producing high-quality schedules that maximize functional unit utilization while at the same time enabling fast instruction issue logic. Many solutions exist to the scheduling problem, ranging from compile-time to run-time approaches. Compile-time solutions feature fast and simple hardware, but at the expense of conservative schedules. Dynamic schedulers produce high-quality schedules that incorporate run-time information and dependence speculation, but implementing these schedulers requires complex circuits that can slow processor clock speeds.

In this paper, we present the Cyclone scheduler, a novel design that captures the benefits of both compile- and run-time scheduling. Our approach utilizes a list-based single-pass instruction scheduling algorithm, implemented by hardware at run-time in the front end of the processor pipeline. Once scheduled, instructions are injected into a timed queue that orchestrates their entry into execution. To accommodate branch and load/store dependence speculation, the Cyclone scheduler supports a simple selective replay mechanism. We implement this technique by overloading instruction register forwarding to also detect instructions dependent on incorrectly scheduled operations. Detailed simulation analyses suggest that with sufficient queue width, the Cyclone scheduler can rival the instruction throughput of similarly wide monolithic dynamic schedulers. Furthermore, the circuit complexity of the Cyclone scheduler is much more favorable than a broadcast-based scheduler, as our approach requires no global control signals.

1 Introduction

In an effort to secure higher levels of system performance, microprocessor designs often employ aggressive scheduling techniques to extract instruction level parallelism (ILP) from serial instruction streams. The goal of the scheduler, be it a compile- or run-time based approach, is to identify and execute independent operations on otherwise unused functional unit resources. By more effectively utilizing functional units resources, instruction throughput increases and program run time is reduced.

As shown in Figure 1, it is possible to implement instruction scheduling at compilation or during execution. Compile-time instruction schedulers analyze the flow of control and data in a program, and then re-order instructions in the binary so that during execution they will better utilize processing resources. The primary advantage of compile-time instruction scheduling is that the hardware necessary to carry out the execution of instructions can be very simple. For example, in machines such as Itanium [8], the compiler selects "bundles" of independent instructions that are fetched and executed as a single unit, obviating the need for any complex dependence checking logic in the underlying hardware.

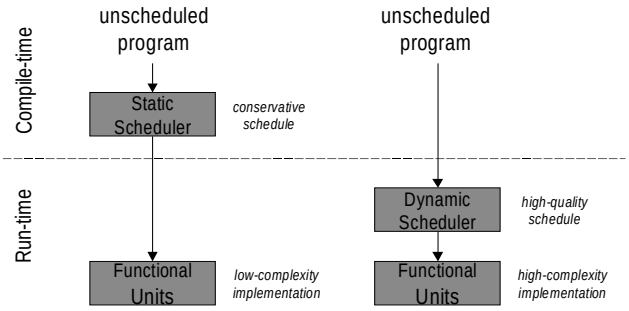

Figure 1. Compile-time vs. Run-time Instruction Scheduling.

The primary disadvantage of compile-time scheduling is that it lacks an accurate assessment of dependencies caused by branches and memory operations, as these are a function of program execution. Consequently, any scheduling decisions implemented in the presence of these instructions must be made conservatively. The conservative tendencies of compile-time schedulers are well recognized, and as such, many proposals have been made to lessen their effect. For example, branch boosting [24] and predication [7] mitigate the effects of control dependencies by pre-computing or hiding branch instructions. Advanced loads [8] and run-time disambiguation [18] reduce the impact of ambiguous load/store dependencies.

Dynamic scheduling at run-time suffers the opposite disposition of compile-time scheduling. Since program dependence analysis occurs at run-time, branch directions and load/store addresses are often available to improve the accuracy of scheduling. Modern designs go another step further and incorporate branch and load/store dependence predictions to further refine the schedules. However, the hardware necessary to implement dynamic scheduling can

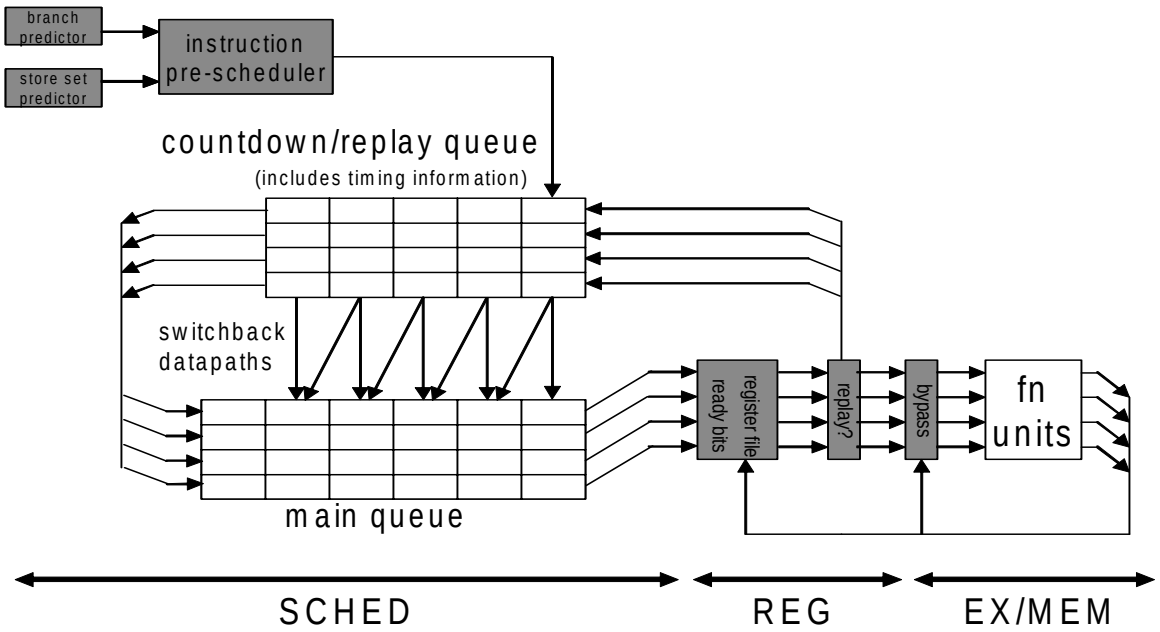

Figure 2. Cyclone Scheduler Architecture

be quite complex and slow. Conventional dynamic schedulers house a "window" of candidate instructions from which ready instructions are sought for execution on available functional units. Instruction windows are implemented using large broadcast-based content addressable memories (CAMs) or dependence-broadcast matrices that track instructions and their input dependencies.

In a dynamic scheduler, it is possible to extract more ILP from a larger instruction window. This increased parallelism, however, will come at the expense of a slower scheduler clock speed, due to an accordingly larger CAM structure. Recent circuit-level studies [19][5] of dynamic scheduler logic have shown that scheduler broadcast logic dominates scheduler circuit performance. As a result, window sizes cannot be increased without commensurate increases in scheduler operation latency. Other recent studies [1] further suggest that as wire latencies grow due to increased deep sub-micron parasitic capacitance effects, these trade-offs may become even more acute with future scheduler designs seeing little benefit from smaller technology sizes.

In this paper we present the *Cyclone* scheduler, a design that draws from compile-time and run-time scheduling techniques in an effort to secure the low-complexity of a compile-time scheduler and the high-quality instruction schedules of a dynamic scheduler. The hardware-based instruction scheduler implements a simple scheduling algorithm similar to compile-time list scheduling [16]. Instructions are scheduled based on the predicted latency until the availability of its operands. To obtain high-quality schedules, we employ our scheduling algorithm at run-time where it is implemented by a simple hardware unit in the front-end of the processor pipeline. To further improve schedules, the hardware-based list scheduler incorporates branch predictions and load/store dependence predictions to speculatively orchestrate execution in the presence of control and memory dependencies. Once an instruction's latency has been predicted, it enters the tail of the Cyclone scheduler queue, which implements a network of locally-synchronized datapaths that route an instruction to its functional unit at (or very shortly after) its proscribed execution time. To ensure correct program execution in the presence of latency and dependence speculation, the Cyclone scheduler incorporates a selective replay mechanism that overloads the register forwarding infrastructure to re-execute only those instructions dependent on incorrectly scheduled instructions. This same mechanism is used to identify and flush instructions which have been squashed due to mispeculation.

The Cyclone scheduler makes three substantial contributions in the area of high-performance dynamic instruction scheduling. They are as follows:

- Broadcast free dynamic scheduling: The Cyclone scheduler contains no global broadcast or control signals. Our design employs distributed methods to schedule instructions, re-synchronize impaired schedules, and recover from mispeculation. The lack of global control enables very fast clocking and lower area due to reduced interconnect requirements.
- Efficient dependence-based variable-latency instruction replay: In the event of a mispredicted latency or dependence, the Cyclone scheduler is capable of recovering schedules by selectively replaying only those instructions dependent on the instruction forcing the replay. The latency of a replay may be variable, and it is set by the instruction that initiated the replay.
- First-class scheduling of memory dependencies: Our design incorporates store-set predictions [4] into the scheduling mechanism and treats store/load dependencies in the same fashion as register dependencies. As a result, our simulations and timing analyses fully account for the impacts of memory communication on the throughput of the scheduler.

The remainder of this paper is organized as follows. Section 2 details the microarchitecture of the Cyclone

scheduler, including implementation of the instruction replay mechanism and the hardware-based list scheduler. Section 3 explains our methodology. Section 4 gives detailed analyses of the Cyclone scheduler's performance. We compare its performance to an idealized monolithic dynamic scheduler and examine the aspects of the Cyclone scheduler that degrade its throughput. We also include detailed analyses of the Cyclone scheduler circuit complexity and area. Section 5 lists related work in scheduler design, Section 6 gives conclusions, and Section 7 suggests future refinements to the Cyclone design.

2 The Cyclone Scheduler

2.1 High-level Architecture

Figure 2 illustrates the high-level architecture of the Cyclone scheduler. After register renaming, instructions enter the *instruction pre-scheduler*, which predicts the number of cycles that must elapse before the instruction's operands are available for execution. Once an instruction's latency has been predicted, it enters at the tail of the *countdown queue*. In the countdown queue, instructions move over locally-synchronized datapaths toward the end (left side) of the queue at the rate of one queue entry per clock cycle. When one half of the predicted latency until execution has expired, instructions jump to the lower *main queue* via *switchback datapaths*. Like the countdown queue, the main queue steps instructions toward execute at the rate of one entry per clock cycle, permitting instructions to arrive at execute at (or near) their predicted execution time.

To facilitate the construction of high-quality instruction schedules, the Cyclone scheduler supports control and data speculation. Load/store dependencies, branch directions and load dependencies are all speculated by the instruction scheduler using a branch predictor and a store-set predictor [4]. If the speculative schedule becomes corrupt (due to, for instance, an input sourced by a load that missed in the data cache, or a late switchback), the Cyclone scheduler is capable of repairing the schedule on-the-fly using a selective replay mechanism that re-executes only those instructions dependent on incorrectly scheduled instructions.

To implement selective replay, instructions access physical storage ready bits immediately before attempting execution. If an instruction arrives at execute and its operands are ready, it commences execution as planned. If the instruction's operands are not yet ready, the latency until execution is once again predicted, and the instruction is re-inserted back into the countdown queue. In addition, the physical register destination of the replayed instruction is marked unavailable in the ready bit table. Subsequent instructions that are dependent on this operation will find this operand unavailable and replay as well. Using this dependence-based replay approach, only those instructions that use the result of an incorrectly scheduled instruction must be replayed. It is important to note that the replay mechanism is sufficiently robust that it can detect and correct any scheduling errors by replaying instructions until their operands arrive. As such, the instruction pre-scheduler need only act as a schedule predictor, any errors it introduces into the instruction schedule will be safely corrected by the replay mechanism.

Wide issue is supported by the Cyclone scheduler by providing additional capacity for scheduling and timing instructions within the Cyclone scheduler queue. It is possible to increase the capacity of the scheduler queues by providing space for multiple instructions in each entry. We term these additional instruction slots *rows*. The scheduler depicted in Figure 2 contains a 4-row Cyclone scheduler queue.

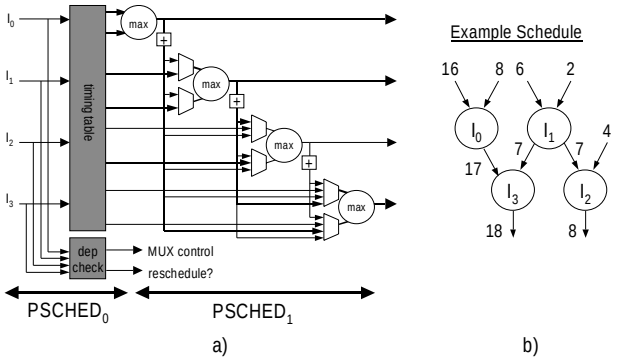

Figure 3. The front-end pre-scheduler architecture is shown in a) and an example chain of four dependent instruction (with delays until execute) are shown in b). All instruction operations in the example execute in a single cycle.

The Cyclone scheduler has lower circuit complexity than conventional broadcast-based scheduler designs (such as a Tomasulo [7] or matrix scheduler [6]) because it lacks centralized control mechanisms of any kind. All communication occurs locally between instruction entries in the Cyclone scheduler queue. Comparatively, broadcast-based schedulers must send dependence information to all other instructions each cycle, making highly capacitive broadcast wires with long propagation delays a necessity.

2.2 Instruction Pre-Scheduler

The instruction pre-scheduler, illustrated in Figure 3a, is responsible for scheduling instructions into the instruction queue such that instructions reach the execute stage immediately after their inputs are computed. Our design is a variant of the data flow prescheduling design presented by Michaud [12]. The instruction pre-scheduler resides in the front-end of the pipeline, after instruction renaming and before the execution queues. Each cycle, instruction placement is computed by the scheduler for each instruction in program order. Instructions are placed at the tail of the countdown queue such that, if an instruction is expected to execute in N cycles, it will be given a timer value that will prompt it to cross over to the main queue in N/2 steps. For a W-wide decode width processor design, the instruction scheduler places the next W dynamic instructions into the tail of the countdown queue each cycle.

Conceptually, the instruction pre-scheduler computes for each instruction the delay until all inputs are available. This task is accomplished with a simple timing table and a MAX calculation. An instruction with two inputs available at times t_0 and t_1 can begin execution at $MAX(t_0, t_1)$. The result of the instruction will be available at $MAX(t_0, t_1)+L$, where L is the latency of the instruction operation. As shown in Figure 3a, each instruction accesses the *timing table*, which produces the delay (in cycles) until the input is available. Each instruction then computes the

maximum delay of its inputs, and the result is used as the index to place the instruction into the queue.

The instruction scheduler timing table is indexed with the logical register index, and it returns the delay until the operand is ready. For instructions with immediate operands or no operands (e.g., load-imm), the timing table returns a zero delay. When an instruction is scheduled, its schedule time is written into the scheduler timing table under the destination register index. To allow the scheduling of dependent instructions in back-to-back fetch groups, the destination delay values are also forwarded back into the pre-scheduler logic.

2.2.1 Limiting Dependence Chains

A complication arises in the design of the pre-scheduler because instructions in an issue group may share dependencies between each other. These dependencies create a recurrence in the scheduler timing computation such that the input to the MAX computations may be the result of the MAX computation of earlier instructions in the same group. We have performed extensive simulations with a range of designs for dealing with inter-instruction dependency scheduling. One conclusion we reached fairly early was that it is insufficient to issue only the independent instructions in each cycle. This insight is not surprising in retrospect, as placing an in-order issue mechanism anywhere before the dynamic scheduler should greatly degrade the throughput of the dynamic schedule.

The scheduler design presented in Figure 3a allows dependent chains of instructions to compute their correct start times within the same cycle. The output of each MAX calculation can be forwarded to any later instruction, to override an input with an earlier computed schedule time. In the first cycle of schedule ($PSCHED_0$), the timing table is probed to determine the delay until input operands are available. In the second cycle of scheduling ($PSCHED_1$), the MAX function computes the time the instruction operands are available, and forwards this results to the inputs of all MAX computations in later cycles.

While the datapaths in the circuit would permit arbitrary length chains of dependent instructions (up to four in the figure), *the dependence logic limits the computation to at most two cascaded MAX computations*. Simulations indicate that dependence chains longer than two instructions happen in less than 3% of all fetch cycles for SPEC2000. Our MAX calculations are limited to 6 or 7 bit subtraction operations (depending on maximum Cyclone queue depth); consequently, the resulting scheduler is both fast and accurate. In the event an instruction chain length is longer than two, the dependence check logic will indicate this case by the end of the $PSCHED_1$ cycle by asserting the *reschedule signal*, which forces an additional cycle to complete the schedule times of long dependent chains. This additional cycle causes very little degradation in performance, since the instructions at the end of long chains would not be ready to execute immediately anyway.

An example of dataflow instruction scheduling is illustrated in Figure 3b. Instructions I_0 and I_1 compute their ready times and forward them to the inputs of instruction I_3's MAX calculation. Instruction I_1's MAX calculation is also forwarded to one input of instruction I_2. After two cascaded MAX calculations, all four instructions have correctly computed their schedule times. It is interesting to note that in a VLIW machine instructions within a fetch group are independent. This would further simplify the instruction pre-scheduler because MAX computations would never need to be forwarded to later MAX computations in the same fetch group.

To keep the pre-scheduler logic simple, it does not accept new instructions until the current fetch group has been fully scheduled. In addition, the instruction scheduler will stall when the queue entry for any of the scheduled instructions is full. In any of these events, the scheduler will retry on subsequent cycles until the full instruction fetch group is scheduled and inserted, at which point scheduling may continue.

2.2.2 Memory Scheduling

It is vital to the production of accurate schedules that loads be scheduled as soon as possible after dependent stores, otherwise, load/store dependencies will cause a significant number of instruction replays. These replays increase the pressure in the Cyclone queue, potentially preventing later instructions from entering the main Cyclone queue, and delaying the execution of loads for a complete trip through the replay loop. To accurately schedule loads, we have adapted the store set dependence predictor [4] to time the execution of load instructions. The store set predictor tracks the stores that are a source for a particular load and assigns to that group a store set identifier. Loads are scheduled after the last store from the store group still in flight when the load is dispatched. The approach can be readily adapted to the Cyclone scheduler by recording with each store set identifier the delay until the last store in flight completes. When scheduling stores, the latency of a store operation is equal to the time to forward a store value to a load instruction, typically the latency for the load/store queue forwarding mechanism (one cycle in our experiments).

2.3 Switchback Datapaths

To implement the instruction schedule computed by the pre-scheduler, 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 toward execution in the main queue. Switchback datapaths provide the connections over which instructions can turn around and head back toward execution. Their inclusion in the design is vital to keeping the scheduler circuit complexity low. With them, it is possible to eliminate any random access ports into the scheduler queue. Instead, all instructions must enter at a single point into the tail entry of the countdown queue, after which they work their way to execution to meet their predicted delay.

Figure 4 illustrates the switchback datapaths and control logic. Instructions time their entry into the main queue using simple countdown logic. Instructions are injected into the countdown queue with a timer value equal to half of their total predicted latency. When the countdown completes the instruction will begin attempting to switchback to the main queue. As shown in Figure 4a, instructions with an even numbered latency jump down to the entry directly below (m0), while instructions with odd numbered latencies jump down and to the left (mp). This routing approach inserts an additional step between entry and execute for instructions with odd numbered predicted latencies. Without the diagonal datapaths, all latencies would have to be even as the same number of queue entries would be traversed down and back to reach execution.

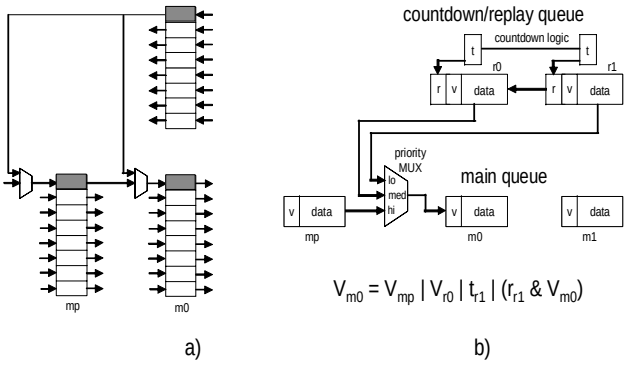

Figure 4. Switchback Logic. The options an instruction has for crossing from the countdown or replay state to the main queue are shown in a). A diagram of this logic for a single instruction, and its critical path logic equation is shown in b).

To keep countdown logic simple, we employ a cascaded Johnson counter [28], which requires N bits to count a maximum of 2N states. The advantage of the Johnson counter is that it only requires one inverter per entry in the countdown queue. An additional feature of the Johnson counter is that we are able to pick counter states such that the most-significant bit of the counter state transitions when switchback should occur, eliminating the need for any comparison logic on counter values.

Because main queue entries can accept instructions from multiple sources (e.g., mp, r1, and r0 in the figure), request conflicts must be resolved. In the event of multiple requests, highest priority is given to the previous main queue entry (mp). This entry must have highest priority because it has nowhere else to go and it cannot stall. If the previous main queue entry is empty, priority is next given to the above switchback channel (r0), followed by the channel diagonally above and to the right (r1). If countdown queue entries cannot enter the main queue due to request conflicts, they will instead enter the following countdown queue entry. This conflict delays the execution of the instruction in the countdown queue by one cycle. Consequently, the instruction will continue to make repeated requests for switchback until either i) it finds an available path, or ii) it reaches the end of the countdown queue at which point it is guaranteed entry into the tail of the main queue. The topology of the Cyclone scheduler queues is such that every instruction will have at least one empty queue entry to move into for the following cycle. In addition, instructions are also guaranteed to eventually reach execution (by traversing the entire length of the queues). As a result, forward scheduler progress is always maintained and no global synchronization is required, which keeps scheduler circuit speeds fast.

Each Cyclone scheduler row is associated with a specific group of functional units. As shown in Figure 4a, instructions switchback to the row in the main queue corresponding to the row they are occupying in the countdown queue. This policy ensures that instructions will find the appropriate functional unit when they reach the execute stage. We found in our circuit analyses that the local nature of switchback control provided significant headroom in circuit performance. As a result, for instructions with multiple functional units available to execute their operations, we permit them to switchback to one of two fixed rows in the main queue (with identical functional units), based on availability of queue space. This policy helps to reduce switchback conflicts and leads to more efficient schedules.

2.4 Replay and Speculation Recovery

The Cyclone scheduler incorporates a unified mechanism to implement selective instruction replay and mispeculation recovery. Immediately proceeding execution, instructions probe a table of physical register ready bits to check the availability of operands. This check is necessary because of the speculative nature of Cyclone scheduling. Any incorrect schedules, due to factors such as latency misprediction, incorrect memory dependence prediction, or switchback conflicts, may alter the schedule, resulting in instructions possibly entering execution before their operands are available. If the ready bits indicate an instruction's operands are ready, its destination register ready bit is set valid and the instruction continues into execution. In the event an instruction cannot complete execution in its predicted execution latency (for example, if a load instruction misses in the cache when a hit was predicted), the instruction sets its destination register ready bit to invalid. Later instructions that access this invalid register will replay and indicate that their result is unavailable, forcing a cascaded dependence-based instruction replay. All instructions that replay enter the countdown queue with a new predicted latency.

Branch and load/store dependence mispeculations are implemented using a similar mechanism. When an instruction behind a mispeculated instruction is squashed, it is marked as such, and the instruction will continue until it reaches execution, at which point the scheduler will drop the instruction. We avoid flushing the scheduler queue as instructions before mispredicted branches may still be in flight, and we also avoid checkpointing Cyclone queue valid bits as this would make access ports into the scheduler queue necessary.

Mispeculated instructions are identified using *speculation masks*, similar to those in the R10000 [29]. The speculation mask of each instruction is included with the instruction in the Cyclone scheduler queues. When instructions reach the end of the main queue, they probe the speculation state. If the table indicates that the instruction has been squashed, it is dropped. For all experiments, we use a five bit speculation mask. This size mask permits at most 32 speculative paths within the window at once. Instructions requiring additional masks would likely be very speculative and have a low probability of retiring.

When a mispeculation occurs, the instruction pre-scheduler timing table must be updated to reflect that instructions squashed no longer are forwarding to instruction being decoded. We leverage two observations to simplify recovery of the pre-scheduler timing table. First, their is no strict requirement that the instruction pre-scheduler timing table be correct. The primary motivation for updating the timing table is that it improves the schedule accuracy for instructions after mispredicted branches. Second, we observe that instructions following a mispeculation nearly always arrive at execute with their operands ready, because in long pipelines it will take many cycles for newly fetched instructions reach execution. During this delay, most instructions in flight will have completed. As an example, simulation of the highly speculative benchmark GCC revealed that 99.98% of all first accessed logi-

cal operands were ready following a mispredicted branch. As such, we can accurately approximate the new pre-scheduler timing table by simply resetting all delay times to zero whenever a mispeculation occurs.

A similar technique is used to recover from load mispseculations. Loads are assigned a speculation mask at decode. In the event a later store arrives after a speculative load completes (revealing an incorrect store-forward), the load's speculation mask and those that follow it are marked invalid and instructions following the load in the Cyclone queue are squashed when they reach execute. Like the register pre-scheduler timing table, all entries in the store-set timing table are set to zero after a mispeculation.

3 Experimental Methodology

3.1 Architecture Simulation

The architectural simulators used in this study are derived from the SimpleScalar/Alpha version 3.0 tool set [2], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

To perform our evaluation, we collected results from all 25 of the SPEC2000 benchmarks [25]. All SPEC programs were compiled for a Compaq Alpha AXP-21264 processor using the Compaq C and Fortran compilers under the OSF/1 V4.0 operating system using full compiler optimization (-O4). The simulations were run for 100 million instructions using the SPEC reference inputs. We used the SimPoint toolset's Early SimPoints [23] to pinpoint program locations to simulate for peak accuracy.

We simulated both the broadcast-based scheduler and the Cyclone scheduler on two different pipeline configurations. First, we simulated a machine with a width of 4 throughout the pipeline, from fetch to commit. Second, we simulated a machine with a width of 8 throughout the pipeline. In addition, we simulated the Cyclone scheduler with an issue width of 8, but leaving the fetch and commit rates at 4 instructions per cycle. The number of functional units was kept constant across all configurations. The processor had 5 integer units, 2 of which were capable of multiplication/division, and 3 FP units, 2 of which were capable of multiplication/division/square root, and 4 memory ports. FU latencies varied depending on the operation, but all FUs, with the exception of the divide units, were fully pipelined allowing a new instruction to initiate execution each cycle.

The Cyclone simulator models the architecture discussed in section 2. The Cyclone main and countdown queues were fixed at at length of 8 which gave the Cyclone an overall loop length of 16. The underlying ROB was varied between 64 and 256 entries. Load/store dependencies are checked within a 32 entry load/store queue.

Our baseline broadcast-based configuration models a current generation out-of-order processor microarchitecture. We modeled machines with instruction window sizes ranging from 16 to 128 and ROB sizes ranging from 64 to 256 instructions. All configurations had a 32 entry load/store queue. There is a 5 cycle minimum branch misprediction penalty.

The memory system for all of the models consists of 32k 4-way set-associative L1 instruction and data caches. The data cache is dual-ported and pipelined to allow up to two new requests each cycle. There is also a 512k 4-way set-associative unified L2 cache with a 12 cycle hit latency. If there is a second-level cache miss it takes a total of 76 cycles to make the round trip access to main memory. The models use a 4k-GSHARE branch predictor with an 8-bit global history and an 2k entry BTB.

In order to reduce the effects of unknown stores and harness dependence information between stores and loads, a store-set predictor [8] is included in both our baseline and our Cyclone configurations. The store-set predictor has 128-entries and is 4-way set-associative. The store-set predictor will find links between loads and sourcing stores allowing loads to speculatively execute before unresolved stores ahead in the load-store buffer.

3.2 Circuit Timing Methodology

To get a full understanding of the consequences of our design decisions, the circuit characteristics of the different scheduler structures must be examined. The circuit delays for the Cyclone scheduler were calculated using the same SPICE design flow we used in [5]. The critical circuit paths were first modeled in SPICE and then optimized by Synopsys's AMPS circuit optimization tool. Finally, timing analysis was performed using Avant!'s HSPICE circuit tool (version 2001.2), using transistor parameters supplied by Taiwan Semiconductor Corporation for their TSMC 0.18μm 1.8V fabrication process. These parameters are available from MOSIS's secure website [15]. The circuit delays for CAM-based scheduler windows used in this study were derived from the models used in [5], as well.

3.3 Area Estimate Methodology

To estimate the chip area of each design, we use the process independent register bit equivalent (RBE) metric defined by Mulder, et al. [17], where one RBE equals the area used by one register file bit. The metric takes into account both the area of the cells themselves, as well as the overhead of control logic, driver logic, and sense amps.

One parameter not accounted for in the original RBE equations given in [17] is the number of access ports for a memory structure. Because the size of each side of a memory bit must scale up linearly with the number of ports, the total effect on area is quadratic [22]. In our model, we apply this port scaling factor to the portions of the RBE area equation that pertain to the footprint of the data bits.

4 Performance Results

4.1 Impact on IPC

The IPC extracted by the Cyclone scheduler is consistently below those produced by the traditional broadcast-based scheduler. The most substantial cause (direct or indirect) of IPC loss is switchback conflicts in the Cyclone. If an instruction wishes to cross from the replay queue to the main queue, it cannot do so if another instruction is currently occupying that slot. This will cause the instruction to cross at a different point which will delay its arrival at the execute stage by at least one cycle. This may not significantly impact the execution of the instruction in

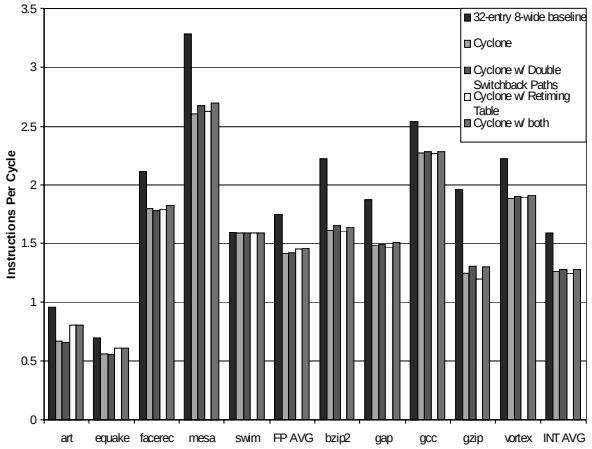

Figure 5. IPC effects of Cyclone optimizations. All configurations have an issue width of 8.

question, but it can disrupt the execution of the descendents of that instruction. When parent instructions fail to execute as scheduled, children that are scheduled for that parent's completion time will arrive early which will force the child instruction as well as all of that child's descendents to replay. Not only are these instructions now forced to replay, but they are also now consuming slots at the tail of the replay queue into which the decode-stage is trying to insert new instructions.

To mitigate these effects, we tried several different approaches. First, because the Cyclone structure scales well with issue width, we made the queues wider in hopes of reducing the number of conflicts. In this case, there were still cases where conflicts would occur due to there being a valid entry in the only row that an instruction was allowed to jump to. Because we determined that the switchback logic was far off the critical path, we gave each entry of the replay queue switchback paths to two different entries in the main queue.

Also, in an effort to improve the accuracy of the Cyclone schedules, we experimented with a Cyclone design which includes variable length dynamic replay. This design adds a retiming table to the register read stage of the pipeline. As instructions leave the main Cyclone queue, they probe the ready bits of their operands. If any operands are unavailable, the retiming table will indicate the number of cycles until the operand is available for use. The instruction then takes the maximum delay of all of its unavailable operands, adds its latency to the result, and then stores this value into the retiming table at the index of its destination register. Finally, the instruction enters the replay queue with a latency equal to the maximum of its unavailable operands. In the baseline system, all replays are signaled with a single ready bit, and the new latency is set to one cycle, meaning that replaying instructions immediately attempt to cross back over to the main queue.

The effects of all these optimizations are shown in Figure 5. Using the double switchback logic, the average IPC was only slightly (~1%) higher, with some benchmarks, like *gzip* seeing as much as 5% improvement. Also, simulation shows that the finer-grained variable-length replay support provided little extra scheduler throughput on most benchmarks. However, *art* and *equake* were 20% and 10% higher, respectively. Given the added area necessary to implement the multi-ported retiming table (approximately 100608 RBEs for an 8-wide configuration), this additional precision is not likely to be worth the complexity cost for a real design.

Table 1. Critical path latencies calculated with SPICE for different scheduler configurations

Config	Timing (ps)	Config	Timing (ps)
Cyclone	193		
16/4-Wide	284	16/8-Wide	345
32/4-Wide	349	32/8-Wide	448
64/4-Wide	466	64/8-Wide	671
128/4-Wide	775	128/8-Wide	1243

Table 2. Component breakdown for scheduler and register file area. Areas are in Register Bit Equivalent (RBE)

Config	Scheduler	Reg File	Total
Cyclone 8-Wide	16682.4	338504	355186.4
CAM 8-W 64-Entry	143527.7	338504	482031.7
Matrix 8-W 64-Entry	58510.1	338504	397014.1

4.2 Circuit Speed

The SPICE circuit timing results are shown in Table 1. The critical path for the Cyclone runs through the prescheduling logic. It consists of two 8-bit MAX operations, and one 8-bit addition. The switchback logic was also simulated, and it was determined to be off the critical path. This was because all communication was only to each entry's neighbors and there was only a small amount of logic. The Cyclone also benefits from not needing any selection logic, since it issues one instruction each cycle from every row.

4.3 Complexity Tradeoff

There are many different factors to take into account when evaluating scheduler designs. The primary goal is high instruction throughput. This can be accomplished either through high IPC or through low-complexity logic which can be run at a higher clock speed. A good scheduler design must also take into account its total power consumption. This factor is tightly related to both the chip area of the design of its circuit complexity.

The Cyclone scheduler takes advantage of this relation by providing a design that is much smaller and less complex than a broadcast-based window. First, because all signals in the structures are local, the throughput is increased due to much faster logic speeds, at the expense of decreased IPC. Also, the Cyclone scheduler structure has a much smaller chip footprint than a broadcast-based scheduler. For example, an 8-decode, 8-issue Cyclone queue takes up approximately 12% of the area of a 64-instruction 8-issue CAM scheduler, and approximately 28% of the area of a similarly sized matrix scheduler.

Both of these advantages are seen in Figures 6, 7, and 8, which analyze the tradeoffs between throughput and area. Throughput is presented in instructions per nanosec-

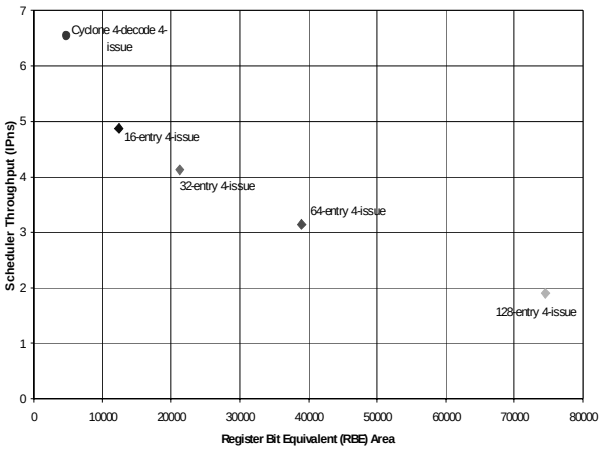

Figure 6. Performance and area for the 4-wide scheduler design space. The most optimal designs are those above (higher performance) and to the left (lower area use) of other designs.

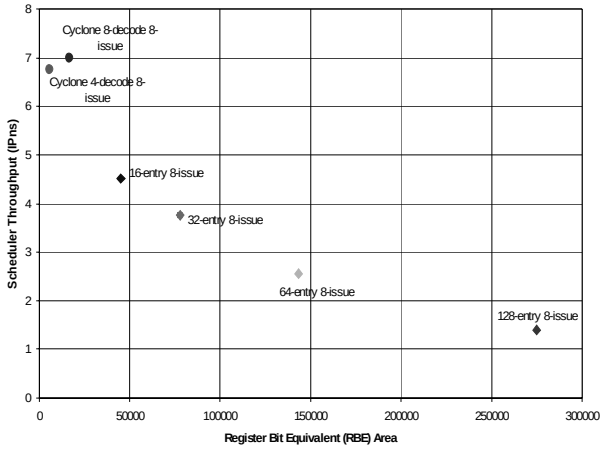

Figure 7. Performance and area for the 8-wide scheduler design space. The most optimal designs are those above (higher performance) and to the left (lower area use) of other designs.

ond (IPns).[1] We examined both broadcast-based and Cyclone-style designs, varying instruction window size (for broadcast designs) and issue width. Figure 6 compares 4-wide execute configurations, while Figure 7 examines 8-wide configurations. Because the size of the register file changes dramatically with the number of ports, its total size is a large factor when making decisions on scheduler design. Figure 8 presents the design space for both widths, with the register file area included. The breakdown of the scheduler and register file area is given in Table 2. The Cyclone scheduler's simplicity and lack of global control allows for fast clock rates and high throughput, but in far less area than same-width conventional designs.

5 Related Work

The technique of timing instruction arrival to execution, based on dynamic dependence information, has been gainfully employed in the past. Our pre-scheduler design is most like Michaud's dataflow prescheduler [12] in which a timed queue structure was filled by a list-like single-pass instruction scheduler. In Michaud's design, instruction's progressed into a small CAM-based dynamic scheduler where there were precisely scheduled. Palacharla's dependence-based FIFO schedulers [19] queues instructions behind dependent operations. The FIFO queues lead to a small CAM-based schedule window. The queues are stepped individually based on availability of operands in the scheduler window. The ILDP processor further refines the dependence-based instruction schedulers to include instruction set support for describing dependent instruction chains [9]. LeBeck's WIB scheduler identifies instructions dependent on long latency operations (data cache misses), and directs these operations to a secondary scheduler [10]. When the long latency operation nears completion, the dependent operations are dumped *en masse* into a small CAM-based dynamic scheduling window. Morancho used a similar approach to move dependent operations following long latency instructions out of the instruction window [14]. Unlike the WIB, they record relative instruction latencies to simplify the re-execution of operations once a valid schedule has been built. We utilize a similar approach in our replay mechanism. As instructions replay, dependencies between dependent operations are maintained by their spacing in the scheduler queues. Unlike the WIB and Morancho's work, our scheduler is completely broadcast free. We pick a schedule and fully commit to it for the lifetime of the instruction, using the replay mechanism to accommodate any incorrectly scheduled instructions. Raasch's segmented instruction queue [20] utilizes course-grained timing information to direct instructions to a sequence of small CAM-based instruction windows. As instructions near execution, they move to instruction queues closer to functional unit results.

Our Cyclone scheduler builds on these previous efforts in a number of ways. First, our design demonstrates that effective dynamic scheduling can be implemented without complex broadcast structures. We build a high-quality preschedule one time and then commit to this decision for the duration of an instruction's lifetime. Broadcast-based schedulers, on the other hand, continually re-evaluate their schedule. Second, our scheduling approach incorporates memory scheduling as a first-class concern in the design. It is, in fact, the great accuracy of the store set predictor that allows our scheduler to make scheduling decisions only once, thus eliminating the need for broadcast structures. Third, our scheduler design incorporates a straightforward mechanism for selective replay of incorrectly scheduled instructions. And finally, our parallel pre-scheduler design reduces logic complexity by recognizing that a vast majority of instructions within fetch groups are independent or form dependence chains of at most two instructions. This observation allows us to efficiently schedule dependent operations within the same cycle.

The use of decentralized dependence analysis (and schedule recovery) is a the central idea of the counter-

1. For the purposes of our study, we assume that the scheduler critical path is the limiting factor in determining the clock speed of the entire processor. This may or may not be true for actual full implementations. Alternatively, if another pipeline stage limits the clock speed gains that can be achieved, the performance headroom afforded by a lower complexity scheduler can instead be used for energy and power savings or to exploit more parallelism.

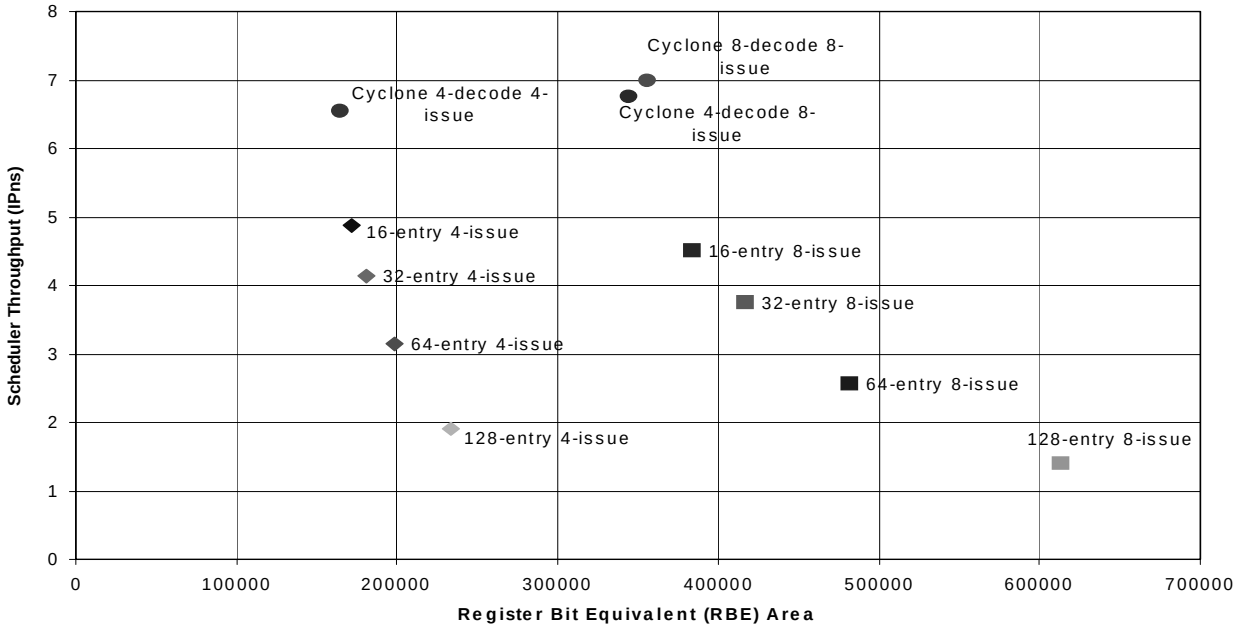

Figure 8. Performance and area overview. Designs shown are for issue widths of 4 and 8 and include register file area. The most optimal designs are those above (higher performance) and to the left (lower area use) of other designs.

dataflow architecture [13][26]. In the counterflow pipeline, instructions and data flow in opposite directions on circular queues. When instructions that are waiting to execute pass input operands, the data needed to execute is captured by the dependent instruction. Once an instruction has captured all its operands, it continues to cycle until it locates an appropriate functional unit, at which time it leaves the queue to begin execution. The approach is similar to our replay queue, which cycles instructions around the replay loop until they meet their input operands. Our design improves on counter-dataflow through the use of pre-scheduling, which is an effective approach to orchestrating the entry of instructions into decentralized dependence queues. More accurate scheduling ensures that when instructions have an opportunity to execute, their operands will very likely be available.

A number of previous efforts have utilized the register forwarding infrastructure to initiate selective instruction re-execution. The sentinel scheduling technique [11] used "poison bits" contained in the register file that were set when load instructions faulted or did not complete. A branch back to the start of the faulting code would then selectively re-execute the faulting code sequence. As instructions read their registers, only those instructions with poison operands needed to re-execute. The approach is quite similar to our replay queue approach, except instead of redirecting program control, we redirect the instructions themselves back into the replay queue. Poison bits were employed in a similar manner by Rogers [21]. The dynamic scheduler was used to identify long latency operations in the WIB scheduler [10]. Using a tagged "pretend ready" register tag broadcast, the design was capable of waking up only the instruction dependent on long latency cache misses.

There have been several other efforts to reduce the complexity of dynamic schedulers. Many current designs bank their selection logic by having separate groups of reservation stations for each functional units [7]. Each of these groups has its own, smaller, selection network. While result tag broadcasts still need to be sent to all of the reservation stations, the latency of selecting instructions for execution is reduced. Stark, Brown, and Patt have proposed two methods for pipelining wakeup and selection logic, allowing for a faster clock. For their first method [27], each reservation station entry carries its own input tags along with its parent instructions' input tags in order to allow back-to-back dependent instructions to execute consecutively. They also propose speculating on which parent instruction will finish last, reducing the number of "grandparent" tags that must be stored. Their second method, select-free logic [3], enables pipelining by allowing all instructions that wakeup to broadcast back into the window the following cycle, even though some of them may not be selected for execution. We observed in our previous work [5] that many instructions have one or more ready operands when scheduled, thus specialized windows can be used to reduce the number of tag comparisons necessary for dynamic scheduling. Tag comparisons were reduced further with a last-tag predictor, which uses reservation stations that ignore all but the tag predicted to be delivered last to the instruction.

6 Conclusions

Traditionally, there have been two primary instruction scheduling mechanisms, both of which have their respective problems. Compile-time scheduling suffers from less than ideal schedules due to lack of information about runtime events. Dynamic schedulers, while able to generate higher quality schedules, utilize very complex issue logic that can become a bottleneck in the instruction pipeline.

We have introduced the Cyclone scheduler, which draws techniques from both these approaches to achieve schedules that rival that of other dynamic schedulers with-

out the need for complex wakeup and selection logic. 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. Even though the Cyclone scheduler design possesses no global communication to slow clock speeds, it rivals the instruction throughput of similarly wide monolithic dynamic schedulers.

7 Future Work

Looking forward, there are a number of additional refinements that could be made to the Cyclone scheduler. Because instruction decisions are made far in advance of actual execution, there exists opportunities to leverage this advanced knowledge of instruction execution patterns. For example, register dependency information could be used to prefetch into a register cache, or instruction opcodes could be use to power-up unused function units. Although not the focus of this work, many optimizations could be implemented to further reduce Cyclone scheduler power. For example, portions of each queue entry could be turned off unless required by execution demands, or dynamic power could be reduced by shifting the issue point rather than the instructions themselves. Finally, because the Cyclone scheduler structure can more easily scale with issue width, it may be applied to some schemes which try to fill up wider pipelines, such as simultaneous multithreading.

Acknowledgements

We would like to thank all of our reviewers and colleagues for their insights and suggestions for strengthening our paper. Specifically, we'd like to thank Mark Brehob for his logic design expertise.

This work was supported the National Science Foundation CADRE program, grant no. EIA-9975286, and by a National Science Foundation CAREER award, grant no. CCR-0093044.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures, In *Proceedings of the 27th International Symposium on Computer Architecture* (ISCA-27), June 2000.

[2] Todd Austin, Eric Larson, Dan Ernst. SimpleScalar: an Infrastructure for Computer System Modeling, *IEEE Computer*, Volume 35, Issue 2, Feb 2002.

[3] Mary D. Brown, Jared Stark, Yale N. Patt. Select-Free Instruction Scheduling Logic, In *Proceedings of the International Symposium on Microarchitecture* (MICRO-34), December 2001.

[4] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th International Symposium on Computer Architecture* (ISCA-25), June 1998.

[5] Dan Ernst and Todd Austin. Efficient Dynamic Scheduling through Tag Elimination, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[6] Masahiro Goshima et al. A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture* (MICRO-34), December 2001.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann Publishers, 2002.

[8] Intel Itanium Architecture Reference Manual, http://www.intel.com/design/itanium/manuals.htm.

[9] Ho-Seop Kim and James E. Smith. An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing, In *Proceedings of the International Symposium on Computer Architecture* (ISCA-29), May 2002.

[10] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses, In *Proceedings of the International Symposium on Computer Architecture* (ISCA-29), May 2002.

[11] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors, *ACM Transactions on Computer Systems*, 11(4):376--408, 1993.

[12] Pierre Michaud and André 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), January 2001.

[13] Michael F. Miller, Kennneth J. Janik, and Shih-Lien Lu. Non-stalling Counterflow Architecture, In *Proceedings of the Conference on High Performance Computer Architecture* (HPCA-4), May 1998.

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

[15] The MOSIS Service, http://www.mosis.com

[16] Steven S. Muchnick. *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, 1997.

[17] J.M. Mulder, N.T. Quach, and M.J. Flynn. An Area Model for On-chip Memories and its Application, In *IEEE Journal of Solid-State Circuits*, Volume 26 Issue 2, Feb 1991.

[18] Alexandru Nicolau. Run-Time Disambiguation: Coming with Statically Unpredictable Dependencies, *IEEE Transactions Computers*, Volume 38 No. 5, May 1989.

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

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

[21] Anne Rogers and Kai Li, Software Support for Speculative Loads, In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '92), 1992.

[22] A. Seznec, E. Toullec, O. Rochecouste. Register Write Specialization Register Read Specialization: A Path to Complexity Effective Wide Issue Superscalar Processors, In *Proceedings of the 35th International Symposium on Microarchitecture* (MICRO-35), November 2002.

[23] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2002), October 2002.

[24] Michael D. Smith, Mark Horowitz and Monica S. Lam. Efficient Superscalar Performance Through Boosting, In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '92), 1992.

[25] Standard Performance Evaluation Corporation, http://www.specbench.org.

[26] R.F. Sproull, I.E. Sutherland and C.E. Molnar. The Counterflow Pipeline Processor Architecture, *IEEE Design and Test of Computers*, Vol. 11 No. 3, Fall 1994.

[27] J. Stark, M. D. Brown, Y. N. Patt. On Pipelining Dynamic Instruction Scheduling Logic, In *Proceedings of the International Symposium on Microarchitecture* (MICRO-33), December 2000.

[28] John F. Wakerly. *Digital Design Principles and Practices*, 3rd edition, Prentice Hall, 2001.

[29] K.C. Yeager. The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, Volume 16, Issue 2, April 1996.