Hierarchical Scheduling Windows

Edward Brekelbaum, Jeff Rupley II, Chris Wilkerson, Bryan Black
Microprocessor Research, Intel Labs (formerly known as MRL)
{edward.brekelbaum, jeffrey.p.rupley.ii, chris.wilkerson, bryan.black}@intel.com

Abstract

Large scheduling windows are an effective mechanism for increasing microprocessor performance through the extraction of instruction level parallelism. Current techniques do not scale effectively for very large windows, leading to slow wakeup and select logic as well as large complicated bypass networks. This paper introduces a new instruction scheduler implementation, referred to as Hierarchical Scheduling Windows or HSW, which exploits latency tolerant instructions in order to reduce implementation complexity. HSW yields a very large instruction window that tolerates wakeup, select, and bypass latency, while extracting significant far-flung ILP.

Results: *It is shown that HSW loses <0.5% performance per additional cycle of bypass/select/wakeup latency as compared to a monolithic window that loses ~5% per additional cycle. Also, HSW achieves the performance of traditional implementations with only 1/3 to 1/2 the number of entries in the critical timing path.*

1. Introduction

The performance of a superscalar microprocessor is a function of its frequency and the amount of Instruction Level Parallelism (ILP) extracted from application code. Several studies, including [2] and [4], have demonstrated that very high single thread ILP is possible. To achieve such parallelism, microprocessors will require larger scheduling windows, higher scheduling bandwidth, and more execution units. Larger windows allow the machine to more easily reach around blocked instructions to find the far-flung ILP in the code sequence. High bandwidth sustains the issue rates required to support a large window, while more execution units enable the execution of more instructions in parallel.

Although very large scheduling windows are effective at extracting ILP, their implementation at high frequency presents three major challenges. First, larger windows imply slower select and wakeup logic. Second, additional execution units present extra load on the bypass network and more distance between units, implying inter-unit delay. Third, large scheduling windows can consume substantial power. Taking into account these design challenges, [11] observed that scaling current scheduler implementations in size, bandwidth, and frequency is becoming very difficult.

This paper introduces a new scheduling mechanism that facilitates the implementation of a very large scheduling window at high frequency. It is based on a hierarchy of scheduling windows. The new hierarchy, referred to as Hierarchical Scheduling Windows or HSW, exploits instructions that are likely to be latency tolerant to reduce implementation complexity. To improve scaling and decrease the impact of size, HSW contains two levels of scheduling windows. The first is a large, slow window and the second is a small, fast window. The slow window provides enormous scheduler capacity for the extraction of far-flung ILP, while the fast window is intentionally small to maintain high frequency. A heuristic is built into the scheduler logic, which implicitly identifies latency tolerant instructions. Latency tolerant instructions are issued for execution from the slow window, while latency critical instructions are issued from the fast window. Each scheduling window has a dedicated execution unit cluster, which simplifies the bypass network. HSW provides a scalable large instruction window that tolerates wakeup, select, and bypass latency, while extracting significant far-flung ILP.

2. Prior Work

There is a significant amount of work in the area of instruction scheduling that studies scheduling algorithms, functional unit cluster effects, bypass networks, and the benefit of identifying critical instructions. Brown et al. [3] described a technique for pipelining the scheduler by removing the select logic from the critical scheduling loop. Combined with a complementary technique proposed by Stark et al. in [14] it is possible to implement a pipelined scheduler by increasing speculation. The special structure of HSW is designed to tolerate the latency of wakeup and select, removing the need for a pipelined scheduler and significantly reducing the amount of scheduler speculation. Palacharla et al. [11] proposed the use of clusters to alleviate the impact of scheduling latency on clock frequency by

distributing the scheduling window and potentially introducing delay between clusters. HSW takes advantage of a different style of clustering that is orthogonal to [11].

The ALU-bypass bottleneck was examined in [1], in which the authors proposed using incomplete bypass. Palacharla et al. also discussed the ALU-bypass problem in [11] and addressed it by implementing another form of deprecated inter-cluster bypass. HSW implements a clustered core that creates latency tolerant bypass logic without deprecating the bypass network performance.

In [5] and [6], Fields et al. found that inter-cluster latency could significantly impact performance and explained how to use criticality information to guide the steering algorithm to mitigate the performance loss due to inter-cluster latencies. HSW does not require a criticality predictor in order to mitigate the performance cost of inter-cluster delay.

In [12], Raasch et al. describes a segmented scheduler based on dependence chains. HSW's implicit heuristic avoids the complexities involved with dependency chain prediction and management. In [16] Hrishikesh et al. describes a simpler segmented scheduler that partitions scheduler bandwidth and weights it according to age. HSW uses age to select instructions for long latency execution in a separate cluster; all bandwidth is fully available throughout the window.

The HSW concept can be used in conjunction with other mechanisms for partitioning the scheduling window; such as distributing the window by functional units, or limiting the mapping between entries in the window and functional units (as implemented in IBM's Power 4 [15]).

3. Hierarchical Scheduling Windows

This section outlines the structure and implementation details of HSW. Figure 1 illustrates the structure of HSW and its four primary components: a slow scheduling window, the register file, a fast scheduling window, and the execution resources with bypass network. The traditional monolithic scheduling window (a single scheduling window) is physically divided into a fast and a slow scheduling window. Each scheduling window has a dedicated and independent scheduler that schedules only instructions within its window. Instructions are dispatched in-order from the front-end of a superscalar machine into the slow scheduling window. From the slow window young ready instructions (latency tolerant) are issued directly to execution cluster #0 and old not-ready instructions (latency critical) are moved to the fast window. Operands are read from the register file as instructions exit the slow window. In the fast window, ready instructions are scheduled by a fast scheduler into Cluster #1.

The HSW implementation exploits instructions that are likely to be latency tolerant to reduce implementation com-

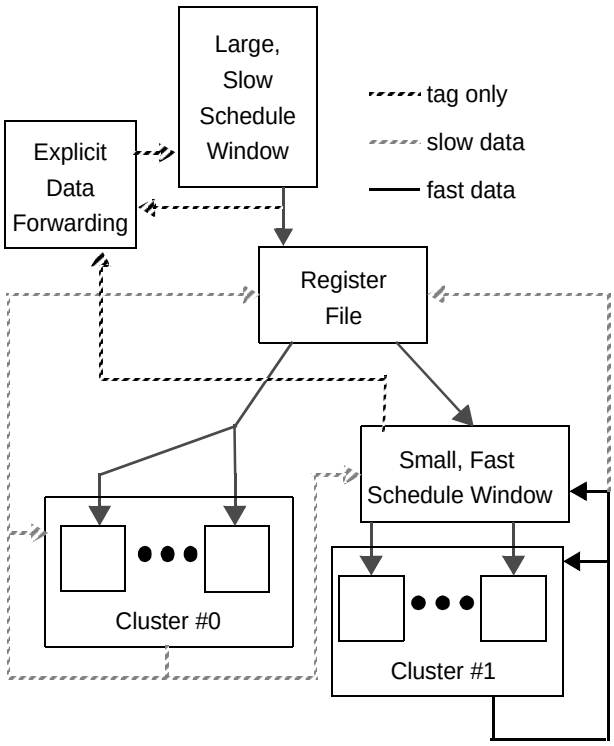

Figure 1. Hierarchical scheduling windows

plexity. The selection heuristics in the slow window identify instructions as either latency tolerant (young ready) or latency critical (old not-ready). Latency tolerant instructions are instructions that can delay execution without impacting performance, while latency critical instructions require immediate execution once they are ready.

The advantage of HSW is the careful allocation of critical resources. The heuristic that moves old not-ready instructions from the slow to the fast window ensures that instructions in the fast window are highly interdependent and latency critical. Limiting the number of instructions in the fast window facilitates a small, fast, data capture implementation. This is optimal for a short schedule loop, which allows simple back-to-back issue of dependent instructions. Issuing only latency critical instructions to the fast window also simplifies the bypass network by dividing it into two regions; a small latency critical network that bypasses data in cluster #1 and a latency tolerant network that services cluster #0 and communication between the two clusters.

On the flip side, having latency tolerant instructions in the slow scheduling window facilitates the implementation of a very large window size, allowing the extraction of far-flung ILP. The slow window can be very large because the latency tolerant instructions it schedules can tolerate extra delay in wakeup, select, and bypass.

The unique structure of HSW renders most components of a large scheduling window latency tolerant including the

majority of the bypass network. The remainder of this section discusses in detail the organization of the execution unit clusters and the slow and fast scheduling windows.

3.1. Execution Clusters

Figure 1 also shows the interaction between the two clusters and the two scheduling windows. Cluster #1 services the fast scheduling window and requires a local tag and data bypass, a writeback path to the register file, and a tag snoop to the slow window for wakeup. Cluster #0 services the slow scheduler and requires a local tag and data bypass, a writeback path to the register file, a tag and data path to the fast window for wakeup and data capture.

HSW is highly tolerant of latency on bypass paths other than the local bypass in Cluster #1. To maintain a rapid back-to-back schedule of dependent instructions, the local bypass in Cluster #1 must be snappy. All other bypass paths can be one or more cycles without significant impact on performance. The latency tolerance of HSW is a virtue of the heuristic used to move instructions from the slow to the fast window. Since the slow scheduler schedules latency tolerant instructions it can tolerate significant latency from execute to subsequent wakeup, and it makes no attempt to schedule dependent instructions back-to-back.

As part of this study the bandwidth and configuration of both clusters are studied, however due to space constraints only the final results can be presented. Each execution cluster has 3 integer, 1 load, 1 store, 1 branch, and 1 floating point unit. The data cache has single load and store ports in all configurations. Three integer units are useful in each cluster to support a 6 wide issue (3-fast, 3-slow). A floating point unit is required in each cluster to support the IPF (Itanium® Processor Family) integer multiply. Unless specified otherwise, all data presented in this paper assume an additional cycle for all result bypass paths, except the critical local bypass in cluster #1 which occurs in the same cycle as execute.

3.2. Slow Window Scheduling

The interaction of the slow window, slow scheduler, and mover is illustrated in Figure 2. The slow scheduler selects young ready instructions for execution in cluster #0, while the mover selects old not-ready instructions for issue to the fast window. An explicit data forwarding block is added to simplify the selection process in the slow scheduler and eliminate content addressable memory (CAM) from the slow window.

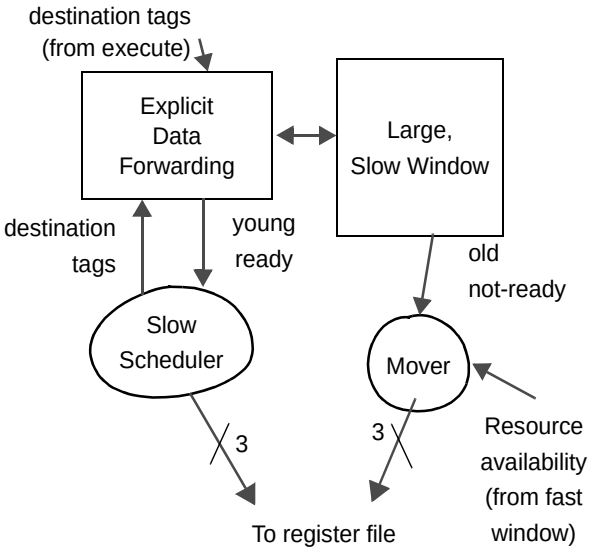

Figure 2. Slow window, scheduler, and mover

3.2.1. Explicit Data Forwarding. As discussed earlier and analyzed in Section 5., the HSW structure exhibits significant tolerance for bypass and select latency in the slow scheduling window. HSW utilizes an implementation of the slow window that takes advantage of this latency tolerance to reduce select and wakeup complexity and eliminate high power CAM logic. Traditionally a large scheduling window requires abundant CAM logic to track incoming tag information and record ready state information. This is not needed in HSW.

Figure 3 depicts an implementation of the slow scheduler adapted from the Explicit Data Forwarding (EDF) system

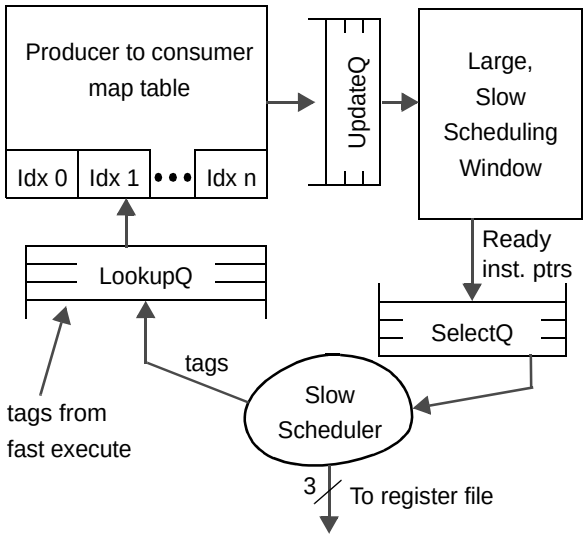

Figure 3. Explicit data forwarding in HSW

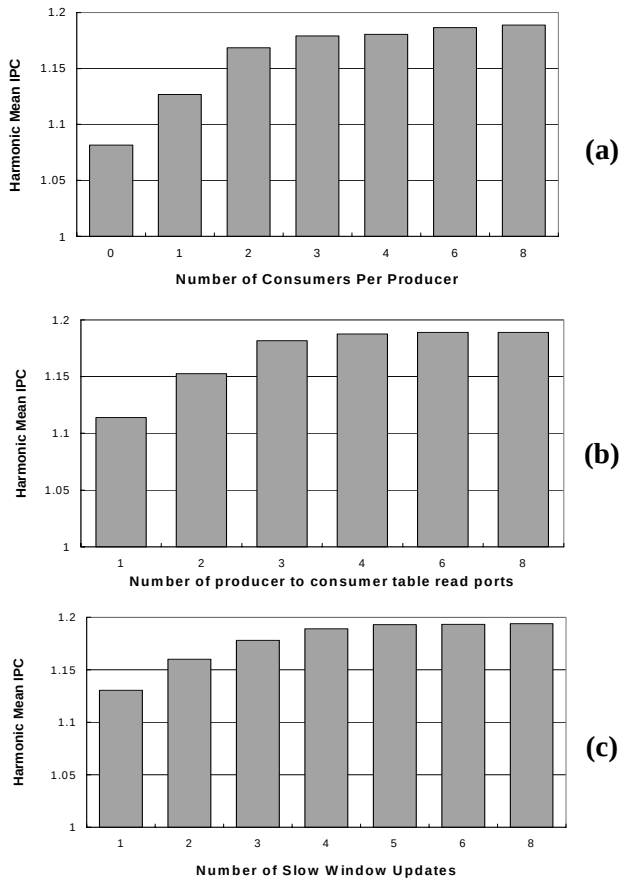

Figure 4. a) Performance impact of consumers recorded per destination tag; b) Performance impact of read ports in the producer to consumer map table; c) Performance impact of slow window update bandwidth.

described in [13]. This mechanism effectively replaces the power hungry CAM logic in a schedule window with a simple RAM lookup. As instructions are scheduled their destination tags are inserted into the LookupQ, where they wait for bandwidth to access the producer to consumer map table. The producer to consumer map table retains a mapping of several dependent instructions per destination tag. Using the Specint2K benchmark suite to evaluate the EDF system, Figure 4a demonstrates that only 6-8 consumers per destination tag are required to maximize performance. Note that a system with 0 consumers is still functional, because instructions ready at dispatch are eligible for slow scheduling. Figure 4b shows that only 4-6 read ports in the map table are necessary to maximize performance. The EDF system is easily implemented at high speed with 6-8 consumers and 4-6 read ports.

After table access, the returned consumer indexes are inserted into an UpdateQ and wait to wakeup the dependent instructions. As shown in Figure 4c only 4 update ports are needed in the slow window. Instructions that are woken up are then inserted into the SelectQ, making them ready for selection. The LookupQ and UpdateQ are used to smooth accesses to the EDF system and the slow window. If any queue overflows, the new entry is simply dropped.

The HSW structure is well suited to the EDF system, because it adds an element of robustness. Should an update and subsequent wakeup be dropped (due to limited resources), the mover will eventually schedule the never-ready instruction for execution in the fast window. The EDF system is obviously multi-cycle. Section 5. explores the performance impact as EDF latency increases.

3.2.2. Slow Scheduler. The slow scheduler speculatively [7][8][10] issues instructions out-of-order from the slow window for execution in cluster #0. Instructions issue speculatively based on resource availability (primarily on the cache ports, if the fast cluster is using a port when a slow instruction arrives, the slow instruction misspeculates and must be rescheduled). Instructions dependent on loads stall until cache hit. To simplify instruction restart, instructions are not removed from the slow window until they are confirmed to be non-speculative. Such a policy reduces the effective size of the slow window, however as explored in Section 5.1., performance is not hindered because the window can be very large.

For large slow window sizes the slow scheduler effectively captures far-flung ILP, substantially improving performance. In the EDF system the slow scheduler is simplified from an out-of-order search of the entire window to a simple in-order read of the first 3 entries in the SelectQ.

3.2.3. Mover. In addition to the slow scheduler a separate mover (Figure 5) can remove instructions from the slow

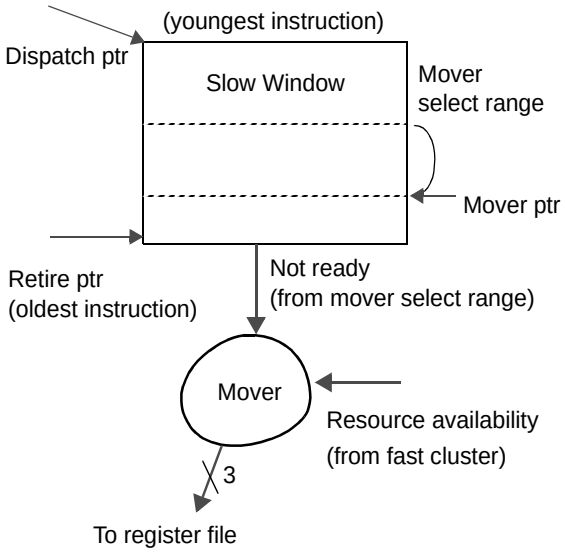

Figure 5. Mover implementation

scheduler window. The mover is a simple scheduler that selects the 3 oldest not-ready instructions from the slow window and copies them to the fast window, provided there is sufficient room available in the fast window. After the mover makes its selection, entries in the fast window are pre-allocated and the instructions are sent to the register file for operand read. The mover heuristic is designed to identify latency critical instructions, which require a fast schedule and execute.

Due to the interaction of the slow scheduler and the mover, the mover is not quite as simple as it appears. Since the slow scheduler selects instructions independently of the mover, it can create fragmentation in the mover's selection window, (where young ready instructions have been issued to cluster #0). Consequently, the next 3 oldest not-ready instructions may not reside in contiguous locations, but instead be dispersed in the slow window. Since the slow window is intended to be very large it is not possible for the mover to search the entire space each cycle. To simplify the search, the mover maintains a head pointer into the slow window, from which it searches only the next 8 instructions. To guarantee forward progress and improve the effectiveness of the mover's small search window, instructions are allocated and deallocated in-order from the slow window.

3.3. Fast Window Scheduling

The fast scheduler, shown in Figure 6., is responsible for scheduling latency critical instructions in the fast window for execution in Cluster #1. Register read is performed before instructions are inserted into the fast window making it a data capture device. (Note: the slow window is prior to register file read and does not require data capture). Therefore, the fast window must capture data from the local bypass. The fast window is intended to be small enough to allow the fast scheduler to schedule dependent instructions back-to-back in a non-speculative fashion.

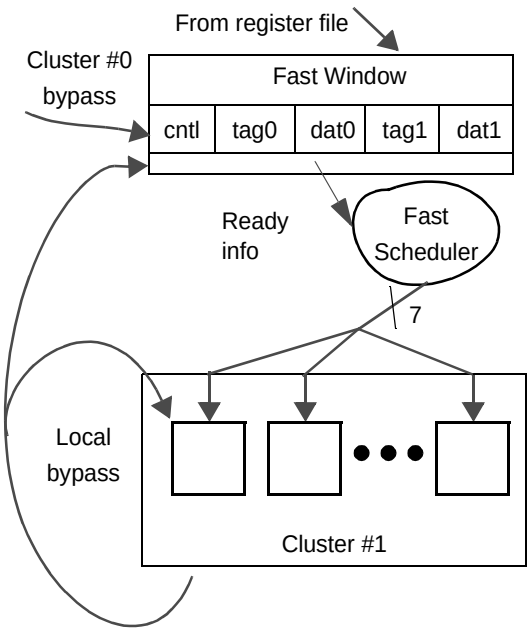

Figure 6. Fast window and scheduler

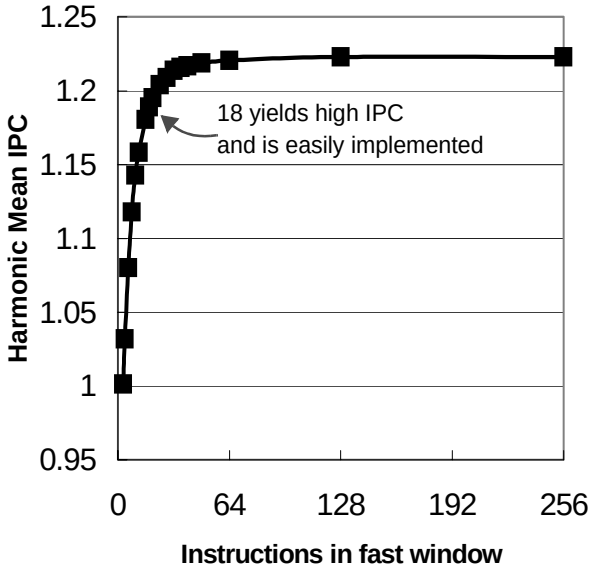

Figure 7. Analysis of the total number of instructions allowed in the distributed fast scheduling window.

As stated earlier the fast window may be implemented any number of ways. After extensive analysis (not documented in this paper) comparing a single window to a distributed window, it is decided that for this study the fast window is best implemented as a distributed window with one out-of-order scheduling sub-window per execution unit in cluster #1, yielding 7 fast sub-windows. To reduce the data storage component of the fast scheduling window, only a limited number of instructions are allowed in the fast window at any one time, regardless of the total capacity of the sub-windows. Figure 7 examines the performance impact of limiting the number of instructions allowed in the fast window, and it appears 18 instructions is an effective and reasonable limit

4. Methodology

This section describes the simulation tool, machine model, and benchmarks used to gather the data presented in Section 5.. This work is based on the Itanium® Processor Family (IPF) architecture and is done within the context of a research project exploring out-of-order implementations of IPF processors. The IPF architecture is an EPIC architecture that utilizes bundles of 3 syllables, instead of instructions. The term instruction is not explicitly defined in the

IPF architecture, however the IPF syllable is comparable to a RISC instruction. For ease of presentation the data presented here is in the form of instructions/cycles or IPC, where an instruction is a single non-NOP syllable that is not predicated false, i.e. we do not count the execution of useless syllables in our IPC measurements.

4.1. Simulation tool

The IPFsim simulation environment is used for all simulation results presented in Section 5.. IPFsim is a research simulation infrastructure for the IPF architecture. IPFsim executes IPF binaries, and contains both a performance model that is cycle accurate and a functional model that provides execution driven capability. All aspects of the architecture and microarchitecture, including the entire memory hierarchy and the main memory controller, are modeled by IPFsim. The next section describes the microarchitecture of the baseline reference machine.

4.2. Machine model

Table 1. outlines the primary microarchitecture parameters. This paper targets a future IPF implementation that may be 5-10 years away with a clock frequency of >10 GHz. Register read is 6 cycles with a total on-chip cache capacity of 17+MBytes. Main memory access times range from 840-2600 cycles with an average of 900 cycles per off chip access. Functional unit latencies are those of the Itanium® 2 microprocessor. This machine model is intended to be very large but not unrealistic for the time frame that it is targeting. The execution core is fully out-of-order[1]. Sufficient renames are provided for all instructions in flight to rename their destinations. The branch predictor is very aggressive with multiple levels of prediction using a neural network [17] (average of the 12 mispredict rates is 3.9%).

4.3. Benchmarks

The SPECint2000 benchmark suite is used for this study. Each benchmark is compiled with the Intel Electron version 6.0 production compiler using peak optimizations. The train input set is used for all simulations. All benchmarks are fast forwarded one billion syllables before cycle accurate simulation begins. Ten two million syllable traces are simulated witha ninety million instruction interval. The cache hierarchy is modeled for all the cycles including the fast forwarding portions.

[1] The simulation environment used in this study addresses all issues related to an out-of-order implementation of the IPF architecture. While they are very interesting, details of this implementation, other than the HSW scheduler, are considered beyond the scope of this paper.

Table 1. Machine model configuration

L1 Instruction Cache	64KB, 16 way 4 cycle latency
Branch Mispredict Penalty	30 cycles
Front-end Width	up to 6 dispatched instructions
Execution Core	256 entry ROB 6 syllables Execution width 6 cycle latency Register File
L1 Data Cache	64KB, 16 way 4 cycle latency 2 ports (1 load + 1 store)
L2	1MB 15 cycle latency 4 requests per cycle (16 banks)
L3	16 MB 45 cycle latency
Memory Latency	840-2600 cycles (DRAM model)

5. Experimental results

This section analyzes the behavior of HSW, confirming claims of efficient implementation and latency tolerance. First, the capacity of the slow scheduling window is examined. Next the slow window's latency tolerance of wakeup, select, and bypass is explored. Last the performance of HSW is compared to more traditional scheduler implementations.

5.1. HSW

By dividing the scheduling window into a two level hierarchy, the HSW structure significantly simplifies scheduler implementation. The bulk of instruction state storage is shifted to the slow window, where wakeup, select, and bypass are extremely tolerant of latency. Figure 8 illustrates the effect the slow window size has on performance, when the fast window is limited to a total of 18 instructions. Impressively, performance continues to increase up to 256 entries. Growing the slow window size from 32 to 256 entry improves IPC by 21.5%, demonstrating the effectiveness of the slow window to extract far-flung ILP.

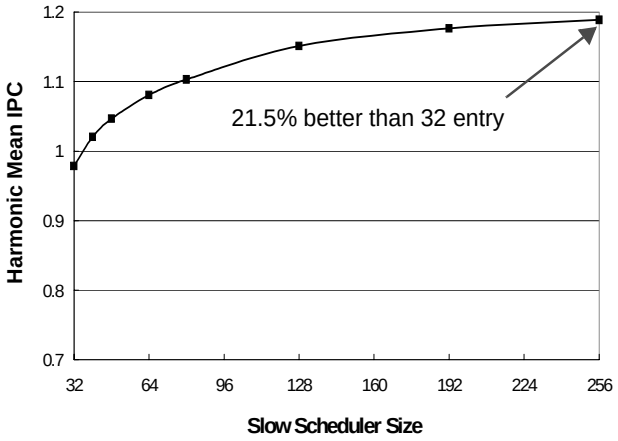

Figure 8. Performance impact of slow scheduler window size

Section 3.2. discusses the slow window tolerance for latency in wakeup, select, and bypass logic. It proposes the use of the multi-cycle EDF system to eliminate all CAM logic in the slow window, reduce bandwidth demands, and facilitate the implementation of the very large slow window, resulting in the tremendous performance in Figure 8. Figure 9 illustrates the performance impact for each additional cycle allocated to the EDF system. As expected the slow window is very tolerant of delay. Each additional cycle costs ~0.5%. Not until the delay is 8 cycles is a significant 3% drop observed.

The combination of a slow and a fast scheduling window is effective because a significant number of instructions can be classified as latency tolerant. If no instructions could be classified as latency tolerant, all instructions would pass through the fast window and the slow window would simply add to the branch penalty and decrease performance. Our data show that ~60% of instructions are executed in the slow cluster; see Figure 10. This data indicates effective utilization of both scheduling windows and demonstrates the effectiveness of HSW.

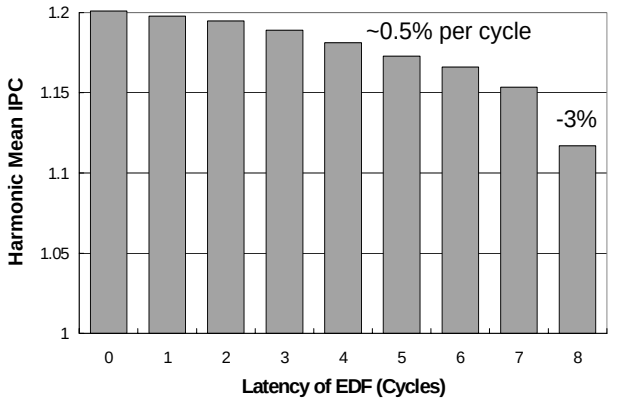

Figure 9. Latency tolerance of the EDF system of the slow window

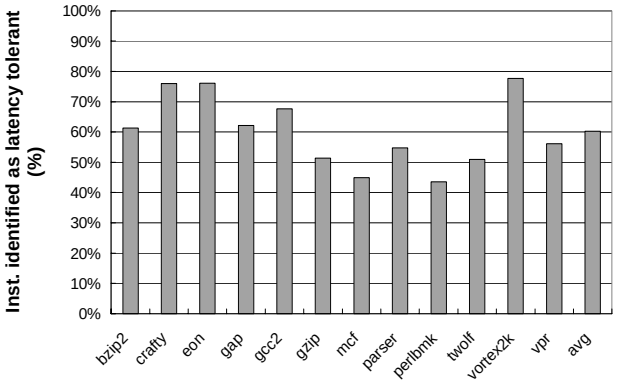

Figure 10. Percentage of instructions executed in slow cluster (#0)

5.2. Comparisons

At this point it is clear that the HSW structure can efficiently implement a very large scheduling window and extract significant performance. This section compares the performance of HSW to three baseline models: A simple monolithic window with speculation, a distributed window with speculation, and a distributed window with a waiting instruction buffer designed to reduce the critical scheduler storage needed for speculative issue.

Figure 11 outlines the monolithic baseline model. The monolithic window is an idealized implementation of a single instruction window with register read after schedule. Up to 9 instructions can be speculatively [7][8][10] selected out-of-order from arbitrary locations in the window each cycle. The execution cluster is similar to a combined cluster #0 and cluster #1. Since a large capacity monolithic window can not be implemented at high frequency, this model is intended as an ideal reference.

The second baseline model is a distributed scheduler, illustrated in Figure 12. A distributed scheduler is a common

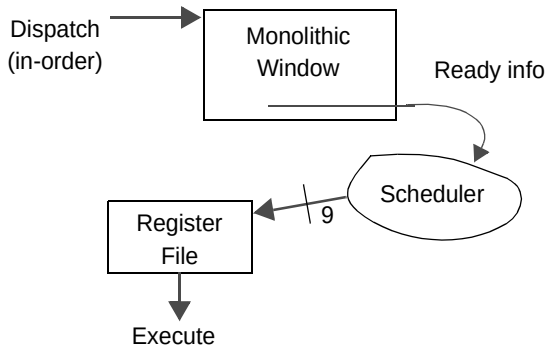

Figure 11. Monolithic scheduler model

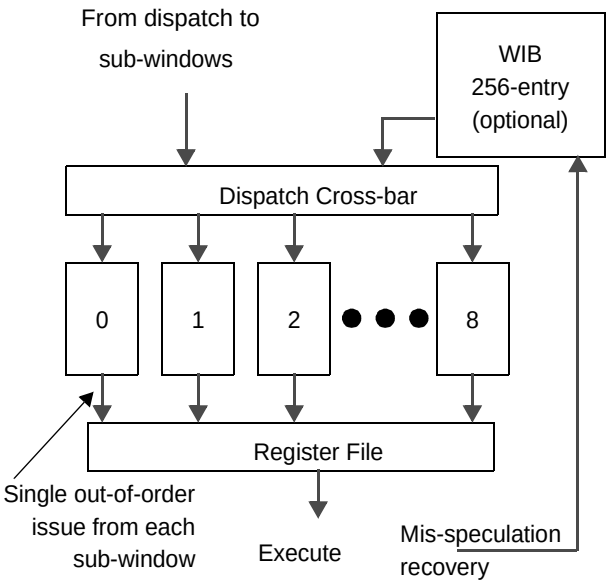

Figure 12. Distributed model (with WIB)

high frequency scheduler implementation [7][8]. This baseline model can evenly distribute instructions across 9 sub-windows, from which 1 instruction is speculatively issued out-of-order from each sub-window per cycle. The execution cluster is similar to a combined cluster #0 and cluster #1.

The third baseline model is the distributed baseline model with the addition of a simplified Waiting Instruction Buffer (WIB) based on [9]. Both the monolithic and distributed baseline models assume speculatively issued instructions remain in the scheduling window until they become non-speculative. The WIB allows instructions to be removed from the window and recovered through the WIB if necessary, thus increasing the effective capacity of the distributed window. The WIB baseline model is more aggressive, but still implementable.

Implementation frequency is critically important when comparing scheduling algorithms. Although not an ideal frequency analysis, it is assumed that the capacity of the largest scheduling window on the critical path will limit the frequency of the scheduling models explored here. It is demonstrated in Figure 9 that the slow scheduling window of HSW is not on the critical path, while the largest sub-window in the fast window is on the critical path. In the distributed and WIB baseline models the critical height is also that of the largest sub-window. The entire capacity of the monolithic model is on the critical path.

Figure 13 compares the performance of HSW to the three baseline models as a function of the number of entries in the critical sub-window. The HSW model has a 256 entry slow window with a limit of 18 instructions across all sub-windows in the fast window. There are 7 sub-windows in the HSW fast scheduler. The distributed and WIB baselines each has 9 sub-windows. As expected the monolithic window requires significant capacity to match the performance of the other models. It is also interesting to note that the WIB model significantly outperforms a traditional distributed model for smaller capacities. HSW consistently out performs all three baseline models including the ideal monolithic model. (Note: the ideal monolithic model is not a “perfect” reference. The true upper bound is a data-capture window at 1.25 IPC) Limiting the HSW fast sub-window to a total of only 6 instructions (noted in Figure 13), HSW achieves performance equivalent to the monolithic scheduler’s ceiling at 256 entries (1.19 IPC). The same HSW model, with a 6 instruction fast sub-window, is 7.7% better than a WIB model with 6 instructions in each sub-window and above the performance ceiling of the distributed model at 1.18 IPC. Finally, Figure 13 demonstrates that HSW achieves equivalent performance of traditional monolithic and distributed scheduler implementations with 1/3 and 1/2 the number of entries on the critical timing path.

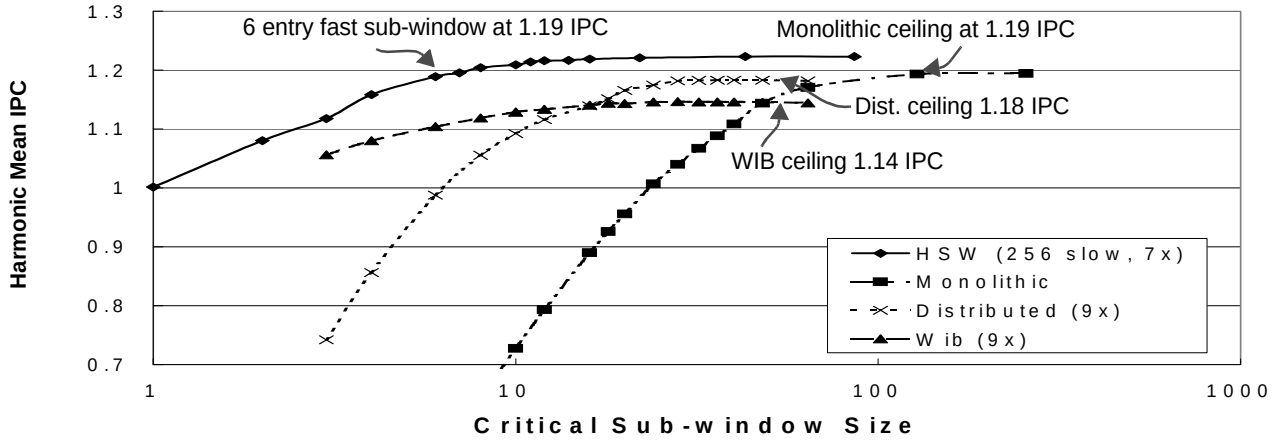

Figure 13. Comparison of HSW to the three baseline models

The primary advantage of the HSW structure is that it performs very well for very small fast sub-window sizes. This is possible because the large, slow scheduling window is effectively extracting far-flung ILP and pulling latency tolerant instructions out of the fast window. Keeping the fast sub-window size small is critically important for latency critical instruction execution. The monolithic, distributed, and WIB models suffer because they require substantially more entries to match the performance of HSW. As the size of each sub-window in the baseline models increases it becomes increasingly more difficult to implement the single cycle wakeup, select, and bypass assumed in Figure 13. Figure 14 compares the latency tolerance of wakeup, select, and bypass of the HSW slow window to the baseline models. An aggressive distributed model with 32 instructions per sub-window and a WIB model with 8 instructions per sub-window are chosen for comparison, because they are at the point of diminishing performance return. Both the distributed model and the WIB model suffer with extra wakeup, select, and bypass latency, losing 17% and 20% performance when 4 cycles are added between dependent instructions, while HSW loses only 1.5%. Clearly, the HSW design not only gives higher performance, it is also more scalable than all three baseline models.

Issue speculation improves performance by allowing back-to-back instruction issue in frequency aggressive machines [7][8][10]. However, excessive mis-speculation can have a negative effect on power consumption and performance. HSW is designed to reduce the amount of issue speculation, thus reducing the negative effects of speculation. This is possible because the fast window schedules mostly dependent instructions, making speculative issue unnecessary. The slow window still speculates on resource availability, but not on load hit, significantly reducing the amount of speculation. Performance is not lost due to the lower speculation rate because the slow scheduler is very tolerant of latency, reducing the need for speculation. Table 2. shows the total number of instruction restarts due to issue mis-speculation in the HSW and the distributed models. On average HSW reduces the number of instruction restarts by 79%, which will have significant impact on power consumption.

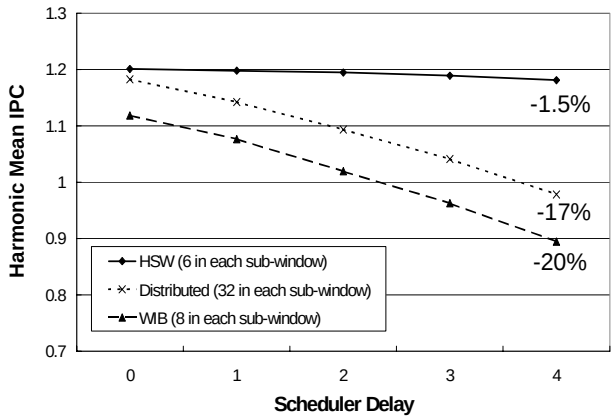

Figure 14. Scheduler latency tolerance comparison

Table 2. Reduction of restart instructions due to issuing mis-speculation

	Total instruction restarts		Reduction
	HSW	Distributed	
bzip2	774,917	1,228,227	37%
crafty	563,332	2,524,339	78%
eon	693,076	1,555,685	55%
gap	1,004,397	3,516,996	71%
gcc2	180,220	1,749,848	90%
gzip	397,634	2,183,146	82%
mcf	173,699	15,516,020	99%
parser	242,623	1,891,218	87%
perlbmk	229,384	1,916,068	88%
twolf	1,986,605	101,154,900	98%
vortex2k	999,681	3,325,361	70%
vpr	150,391	3,038,283	95%
average	616,330	11,633,341	79%

6. Conclusion

It is shown that the identification of latency tolerant instructions can be effectively exploited to construct the unique structure of HSW. The organization of HSW simplifies scheduler implementation, rendering most components of a large scheduling window very latency tolerant, including wakeup and select logic, and the majority of the bypass network.

On average 60% of instructions are identified as latency tolerant, enabling the slow scheduling window to extract significant far-flung ILP, while losing only 0.5% performance per additional cycle of wakeup, select, and bypass latency. It is also shown that HSW uses available resources more efficiently, achieving equivalent performance of traditional monolithic and distributed scheduler implementations with 1/3 and 1/2 the number of entries on the critical timing path. The HSW structure also reduces the number of instruction restarts by 79%, which will have significant impact on power consumption. It is clear that HSW achieves higher performance and is more scalable than traditional scheduling methods.

The HSW design exploits the observation that ~60% of the instructions are latency tolerant and need not be executed as fast as possible. This observed attribute is related to the instruction criticality and vitality that others have observed [5]. However, this paper focuses on leveraging this attribute to implement an efficient large scheduling window. There is no explicit hardware in HSW for identifying critical instructions. Latency tolerance is implicitly identified by young instructions that are ready in the slow window. This paper makes no attempt at a detailed comparative study of instruction latency tolerance vs. instruction criticality or vitality. This is the subject of future work.

7. References

[1] P. Ahuja, D. Clark, and A. Rogers, "The performance impact of incomplete bypassing in processor pipelines." In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 36-45, November 1995.

[2] T. Austin and G. Sohi. "Dynamic dependency analysis of ordinary programs." In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 342-351, June 1992.

[3] M. Brown, J. Stark, and Y. Patt. "Select-Free Instruction Scheduling Logic." To appear in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[4] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. "Single instruction stream parallelism is greater than two." In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 276-286, May 1991.

[5] B. Fields, S. Rubin, and R. Bodik. "Focusing Processor Policies via Critical-Path Prediction." In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 74-85, June 2001.

[6] B. Fields, R. Bodik, M. Hill. "Slack: Maximizing Performance Under Technological Constraints." In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 47-58, May 2002.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. "The Microarchitecture of the Pentium® 4 Processor." http://developer.intel.com/technology/itj/q12001/articles/art_2.htm, Intel 2001.

[8] R. Kessler. "The Alpha 21264 Microprocessor." In *IEEE Micro*, pp. 24-36, March 1999.

[9] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. "A Large, Fast Instruction Window for Tolerating Cache Misses". In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 59-70, May 2002.

[10] E. Morancho, J. Llaberia, and A. Olive. "Recovery Mechanism for Latency Misprediction." In *Proceedings of the International conference on Parallel Architectures and Compilation Techniques*, pp. 118-128, September 2001.

[11] S. Palacharla, N. P. Jouppi, and J. Smith. "Complexity-Effective Superscalar Processors" In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206-218, June 1997.

[12] 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*, pp. 318-329, May 2002.

[13] T. Sato, Y. Nakamura, and I. Arita, "Revisiting Direct Tag Search Algorithm on Superscalar Processors". In *Workshop on Complexity-Effective Design held in conjunction with the 28th Annual International Symposium on Computer Architecture*, June 2001.

[14] J. Stark, M. Brown, and Y. Patt. "On Pipelining Dynamic Instruction Scheduling Logic". In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pp. 57-66, December 2000.

[15] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. "POWER4 System Microarchitecture". http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.pdf, IBM, October 2001.

[16] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, P. Shivakumar. "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays". In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 14-24, May 2002.

[17] D. Jimenez, C. Lin. "Dynamic Branch Prediction with Perceptrons". In *Proceedings of the 7th Annual International Symposium on High Performance Computer Architecture*, pp. 197-206, January 2001.