Dynamic Scheduling with Narrow Operand Values

Erika Gunadi

University of Wisconsin-Madison

Department of Electrical and Computer Engineering

1415 Engineering Drive

Madison, WI 53706

Submitted in partial fulfillment of the M.S. Degree in Electrical and Computer Engineering

Project Option

June 2005

Dynamic Scheduling with Partial Operand Values

Abstract

Tomasulo's algorithm creates a dynamic execution order that extracts a high degree of instruction-level parallelism from a sequential program. Modern processors create this schedule early in the pipeline, before operand values have been computed, since present-day cycle-time demands preclude inclusion of a full ALU and bypass network delay in the instruction scheduling loop. Hence, modern schedulers must predict the latency of load instructions, since load latency cannot be determined within the scheduling pipeline. Whenever load latency is mispredicted due to an unanticipated cache miss or store alias, a significant amount of power is wasted due to incorrectly issued dependent instructions that are already traversing the execution pipeline.

This paper exploits the prevalence of narrow operand values (i.e. ones with fewer signficant bits) to solve this problem, by placing a fast, narrow ALU and datapath within the scheduling loop. Virtually all load latency mispredictions can be accurately anticipated with this narrow data path, and little power is wasted on executing incorrectly scheduled instructions. We show that such a narrow data-path design, coupled with a novel partitioned store queue and pipelined data cache, can achieve a cycle time comparable to conventional approaches, while dramatically reducing misspeculation, saving power, and improving per-cycle performance. Finally, we show that due to the rarity of misspeculation in our architecture, a less-complex flush-based recovery scheme suffices for high performance.

Keywords: scheduler, issue-queue, partial operands, microarchitecture.

1 Introduction and Motivation

Over the last two decades, microprocessors have evolved from relatively straightforward, pipelined, largely non-speculative implementations to deeply pipelined machines with out-of-order execution and a high degree of speculation to maximize performance benefit. One technique that is commonly implemented in current generation designs is load latency speculation. In this technique, the scheduler speculates on load latency by assuming no store aliasing and cache hits, and then issues dependent instructions speculatively to avoid bubbles in the execution schedule. As the pipeline depth between the scheduling stage and the execution stage increases, this technique becomes increasingly important for exposing high degrees of instruction-level parallelism.

Unfortunately, this speculative scheduling technique comes with added complexity and power. All speculative techniques require mechanisms to recover from misspeculation and guarantee correctness. Many different techniques have been proposed and implemented over the last few years; they come with various tradeoffs between complexity and accuracy. The least complex and the least accurate technique is refetch recovery that uses the same mechanism as branch misprediction recovery. The most complex and the most accurate is parallel selective recovery, where only instructions dependent on the latency-mispredicted load are recovered and further propagation of incorrect speculative wakeup activity is terminated immediately. Less complex solutions are employed in existing machines, such as serial selective recovery in Pentium 4 [1] or squashing replay in Alpha 21264 [2]. Selective recov-

Table 1: Percentage of Replayed Instructions

Integer Benchmark	2-Cycle Store to Load Forward Latency			3-Cycle Store to Load Forward Latency		
	Parallel Selective Replay	Serial Selective Replay	Squashing Replay	Parallel Selective Replay	Serial Selective Replay	Squashing Replay
bzip2	0.56%	3.47%	1.91%	0.58%	3.50%	2.13%
crafty	2.48%	7.53%	10.87%	4.94%	11.45%	15.35%
eon	1.06%	3.69%	8.99%	11.55%	27.30%	30.23%
gap	0.39%	1.63%	2.15%	1.11%	3.84%	3.66%
gcc	0.96%	2.32%	3.31%	4.22%	11.71%	9.03%
gzip	2.90%	6.04%	8.40%	8.18%	25.41%	14.14%
mcf	10.61%	22.63%	30.30%	11.10%	31.26%	96.20%
parser	3.39%	9.07%	9.00%	7.03%	15.99%	16.63%
perlbmk	0.46%	0.93%	6.11%	2.56%	5.98%	14.28%
vortex	2.32%	4.43%	12.03%	7.76%	15.96%	27.67%
vpr	5.66%	13.61%	18.73%	8.04%	24.49%	42.03%

ery precisely replays dependent instructions, but fails to stop the propagation of incorrect wakeup activity. Squashing recovery ,replays all dependent and independent instructions issued in the shadow of misspeculated load. However, a parallel selective recovery scheme where an instruction can cause a recovery event at any point during execution is barely feasible in current generation designs and is not scalable to future machines that are likely to be wider and deeper [3].

Another important aspect to consider in speculative techniques is power. As the degree of misspeculation increases, more power is wasted to execute misspeculated instructions and to re-execute them when the misspeculation is resolved. Table 1 illustrates the percentage of replayed instructions over executed instructions for the SPEC2000 integer benchmarks. The numbers in Table 1 are generated using the machine configuration described in Section 5. These percentages correlate with the percentages of dynamic power wasted in the out-of-order core due to load latency mispredictions. The three different replay schemes shown are parallel selective replay, which is the most complex and accurate scheme, serial selective replay similar to the one used in Pentium 4, and squashing replay used in Alpha 21264. Two different configurations are shown in Table 1. The first configuration uses the same latency for store-to-load forwarding and data cache hits so that store-to-load forwarding does not cause latency misspeculation. The second configuration uses a store-to-load forwarding latency that is one cycle longer than data cache hit latency so that a store-to-load alias causes load-dependent instructions to replay (such a latency mismatch causes replays in the IBM Power 4 [4]). As can be seen from the table, the amount of wasted activity due to misspeculation is quite significant under serial selective replay and squashing replay.

The power issue becomes more important as clock rates and die sizes increases. Already, current generation designs are reaching the limits of conventional air cooling. As clock rates continue to increase, power dissipation soon becomes a key limitation for microprocessor performance improvement techniques. Since the out-of-order core

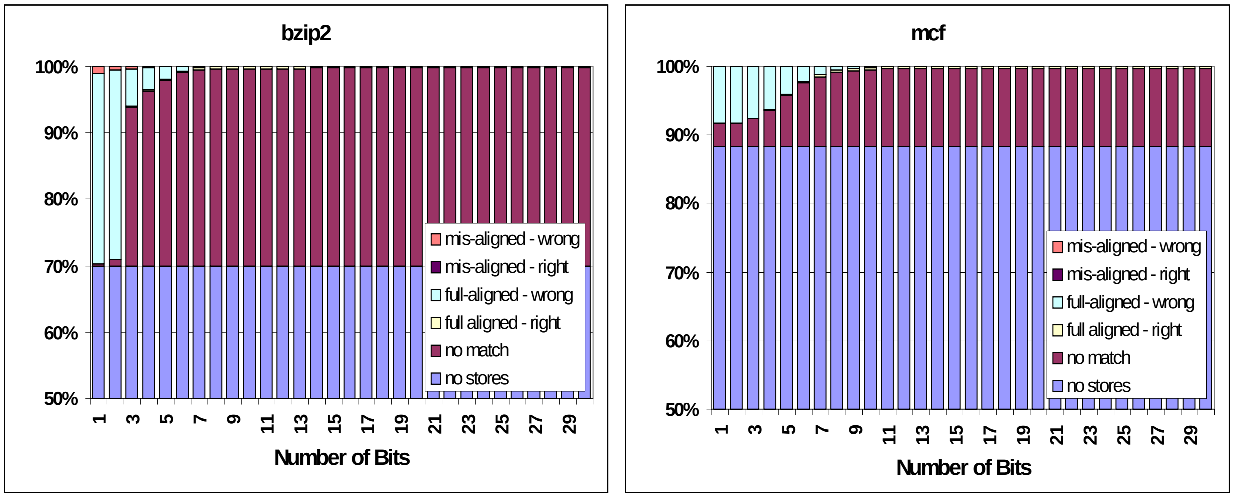


FIGURE 1. Percentage of Load-Store Disambiguation per Number of Bits. No stores refers to no stores in front of the load. No match refers to no store mach in lsq. Full-aligned refers that the partial bits predict a prior fully-aligned store. Mis-aligned refers to partial bits predict a non-fully-aligned stores in lsq.

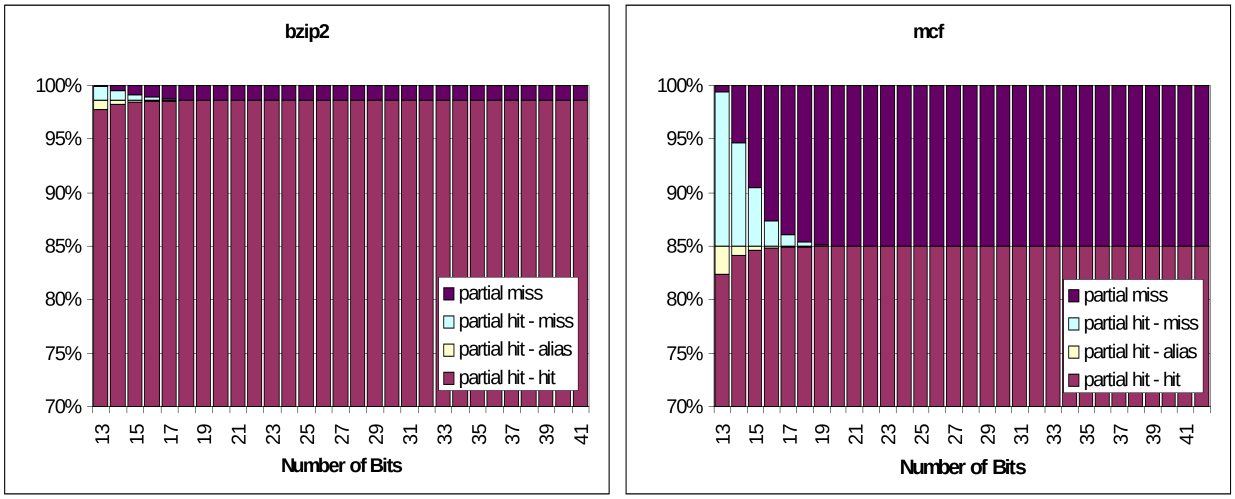


FIGURE 2. Percentage of Partial Tag Match per Number of Bits. Partial hit - hit means that partial bits predict a load to be and the prediction is right. Partial hit - alias and partial hit - miss refers to a cache hit prediction that is wrong due to associative aliasing and cache miss. Partial miss refers to cache miss that is predicted using partial bits

consumes about 40%-50% of total chip power [5], reducing the number of replayed instructions is an attractive target for microarchitectural optimization.

In this paper, we propose a speculative scheduler that uses partial operand knowledge to reduce the occurrence of load latency misspeculation. It is widely known that full operand bits are not necessary to do load-store disambiguation or to determine cache misses [6][7]. Figure 1 illustrates that in most cases, after examining the least significant ten bits of addresses, a unique forwarding address is found or all addresses are ruled out, allowing a load to pass prior stores. Figure 2 similarly shows that a large percentage of cache hits (tag matches) and misses can be deter-

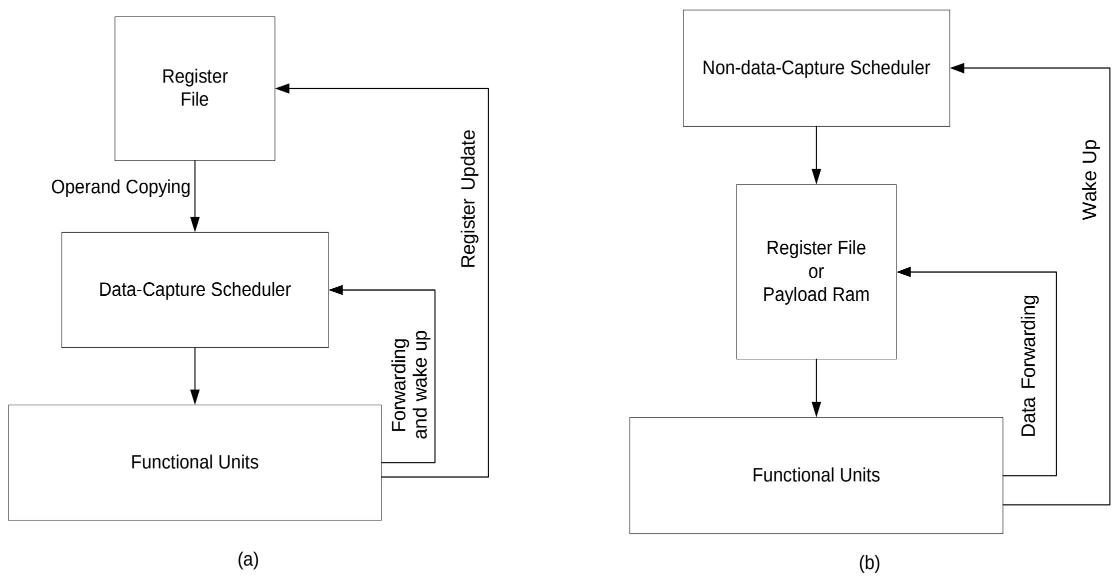


FIGURE 3. Data Capture Scheduler (a) and Non-data Capture Scheduler (b).

mined by using the first 15-20 bits of the address tags. In fact, the Pentium 4 processor addresses its level-one data cache using only 16 bits of the virtual address. We propose further exploiting this phenomenon by capturing several bits of the operands in the scheduler and using them to help with load latency prediction. Our analysis shows that capturing some number of bits in the scheduler does not really affects the scheduler latency relative to a baseline speculative scheduler of the same capacity. We also show that we can reduce the power consumption in the out-of-order core by dramatically reducing load misspeculation, and also by simplifying the store queue and data cache access paths. Lastly, we show that we can reduce the complexity of the scheduler by using a less aggressive recovery mechanism, while sacrificing very little performance.

The rest of the paper is structured as follows. Section 2 describes and compares different kind of schedulers. Section 3 describes the details of partial data-capture scheduler and how it is integrated with the rest of the pipeline. Section 4 explains the modeling of the scheduler, execution pipeline power consumption, and microarchitectural model. Section 5 provides detailed performance and power evaluation of our partial data-capture scheduler, and Section 6 concludes the paper.

2 A Brief Scheduler Overview

Based on the location where the operands are stored, schedulers can be divided into two categories: data capture schedulers and non-data capture schedulers [9]. As is clear from the name, data-capture schedulers store the operands in the scheduler itself while non-data capture schedulers store them either in a physical register file or a separate payload RAM. Speculative scheduling must be introduced to enable back-to-back scheduling of dependent instructions once the number of cycles to access the register file or payload RAM exceeds zero.

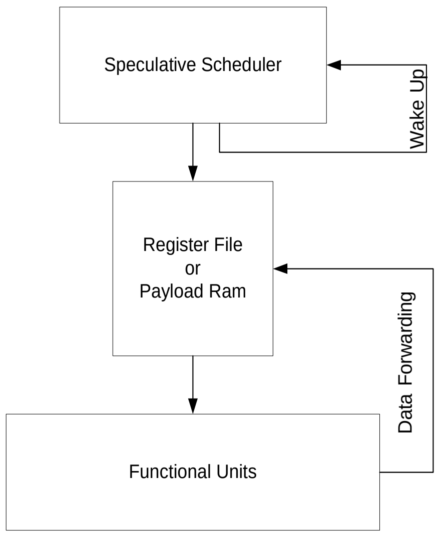


FIGURE 4. Speculative Scheduler.

2.1 Data Capture vs Non-Data Capture Scheduler

As mentioned before, a data capture scheduler stores the operands in the scheduler itself. Figure 3(a) illustrates a scheduler with data capture. In this scheduler design, operands that are ready are copied from the register file (either architected or physical) into the scheduler when dispatching instructions. For the operands that are not ready, tags are copied into the scheduler and used to latch in the operands when they are forwarded by the functional units. Results are forwarded to the waiting instructions in the scheduler. In effect, result forwarding and instruction wake up are combined in a single physical structure.

In contrast, non-data capture schedulers do not store the operands in the scheduler itself, as illustrated in Figure 3(b). In this style, register read or payload RAM access is performed after the scheduler, as instructions are being issued to the functional units. At instruction dispatch there is no copying of operands into the scheduler; only tags for operands are loaded into the window. The scheduler still performs tag matching to wake up ready instructions. However, results from functional units are only forwarded to the register file and/or payload RAM. In effect, result forwarding and instruction wake up are decoupled.

The main advantage of a non-data capture scheduler over a data capture scheduler is the possibility to achieve a faster cycle time since operand write and read operations and ALU execution are not done in the same cycle as scheduling. However, additional cycles needed to access the operands before going to the functional unit effectively delay the scheduling of dependent consumer instructions, and prevent them from executing in a back-to-back fashion.

2.2 Speculative Scheduling

A speculative scheduler is a natural solution for re-enabling back-to-back execution of producer-consumer instructions in a non-data capture scheduler. As the number of cycles needed to access the register file or payload

RAM increases, it becomes too expensive to wait for the producer to finish executing before issuing its consumers. Thus, dependent instructions are issued based on the predicted latency of their parent instructions. Since all functional unit latencies other than memory latency are constant, it is quite easy to predict the latency of an instruction and schedule its dependents accordingly. As illustrated in Figure 4, a speculative scheduler decouples the wake up loop from the execution loop to enable back-to-back execution.

The obvious benefit of speculative scheduling is its ability achieve higher instruction-level parallelism, hence better performance. However, this technique comes with added complexity and power due to speculation and recovery, as mentioned in Section 1.

3 Dynamic Scheduling with Narrow Operand Values

We propose a scheduler that exploits partial operand knowledge to help predict load latency speculation. A subset of the operands, sixteen bits, are stored in the scheduler and used to resolve load-store aliases and perform partial tag matching [8] on the cache. In this section, we explain the details of our scheduler and how to integrate that scheduler into the rest of the pipeline. Detailed cycle time, power, and area analysis for our proposed design is presented in Section 3.3.

3.1 Scheduler with Narrow Data-Path

Our scheduler tries to exploit partial operand knowledge by storing the least significant sixteen bits of the operand inside the scheduler itself. We decided to use only the least sixteen bits of the operands since, as illustrated by Figure 1and Figure 2, these sixteen bits are sufficient fo load-store disambiguation and partial tag matching in the data cache for the vast majority of load instructions.

Figure 5 (a) shows a naive implementation of a scheduler with a narrow data path. Narrow ALUs are added in the scheduling stage to generate data to be forwarded to dependent instructions. A narrow store queue access port and a narrow cache access port is added to see whether a load has a store alias or whether it misses the cache. In addition to a tag broadcast bus, narrow data broadcast buses are also added to forward the data back to the scheduler. Unfortunately, complex integer operations such as division, multiplication, and right shift have to schedule their dependent instructions non-speculatively since it is difficult to break such functional units into narrow and wide units.

In contrast to a non-data capture scheduler, our narrow data capture scheduler has two possible critical paths. As can be seen from Figure 5, the first path, shown with a solid line, is the tag broadcast loop. The second path, shown with a dotted line, is the data broadcast loop. Based on Figure 5, the likely critical paths are:

Non-data capture scheduler:

select - mux - tag broadcast and compare - ready signal write

Narrow-data-capture scheduler:

select - mux - tag broadcast and compare - ready signal write (1)

select - mux - narrow ALU - data broadcast - data write (2)

Both the select logic and narrow ALU contribute substantial delay, so the second delay path is most likely to be the critical path in this kind of scheduler. The fact that there are a limited number of narrow ALUs prevents us from accessing the ALU in parallel with the select logic. However, by trading additional area for reduced delay, the narrow

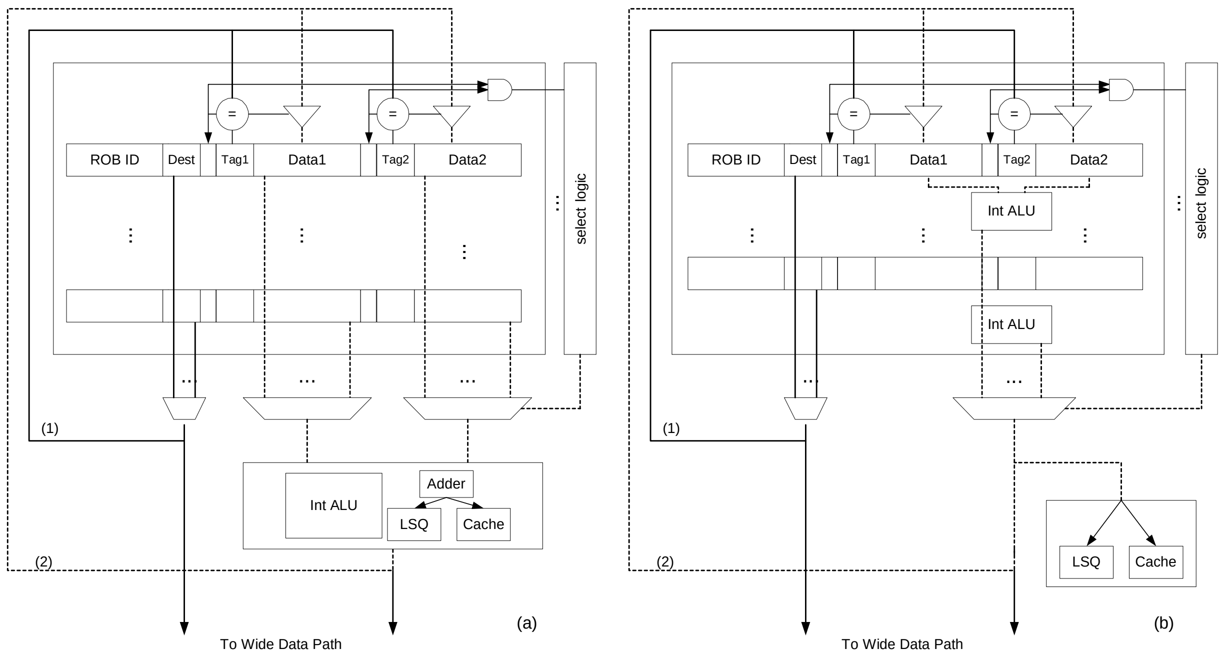


FIGURE 5. Scheduler with Narrow Data-Path (a) and Scheduler with Embedded Narrow ALUs (b).

ALU latency can be reduced or even eliminated for integer ALU instructions, by embedding narrow ALUs in every scheduler entry.

Figure 5 (b) shows narrow data capture scheduler with narrow ALUs embedded in every scheduler entry. In this scheduler, narrow integer ALU execution is done in parallel with the select logic, hence reducing the second path into: *max(select, partial ALU) - mux - data broadcast - data write*. Using only sixteen bits of partial data, we expect that the partial ALU latency will not exceed select logic latency. Therefore, the latency of the second path should not differ considerably from the latency of the first path, if at all.

Astute readers may have noticed that the improvement gained by adding ALUs will not noticeably improve the latency of load instructions, since store queue access and cache access have to wait for the select operation. Fortunately, load-dependent instructions do not need to be issued immediately after the load is issued. Since a load takes more than one cycle to execute, the partial load execution unit does not need to schedule its dependents until some later cycle. Since partial cache access and partial store queue access need less time than a full cache access and full store queue access, we can easily complete these actions in time to wake up load-dependent instructions despite the additional select delay.

3.2 Pipelined Data Cache

In order to enable cache access using partial data, our scheme utilizes a pre-existing technique named partial tag matching. We physically partition the cache into two separate banks to avoid adding more ports for the partial access. The first narrow bank is only used for partial access, while the wide bank is used to access the rest of the tag and data. Since partial addresses are available early, we employ a technique similar to [10] to activate only a single subarray in the wide bank, conserving substantial decoder and subarray access power. We do not employ this tech-

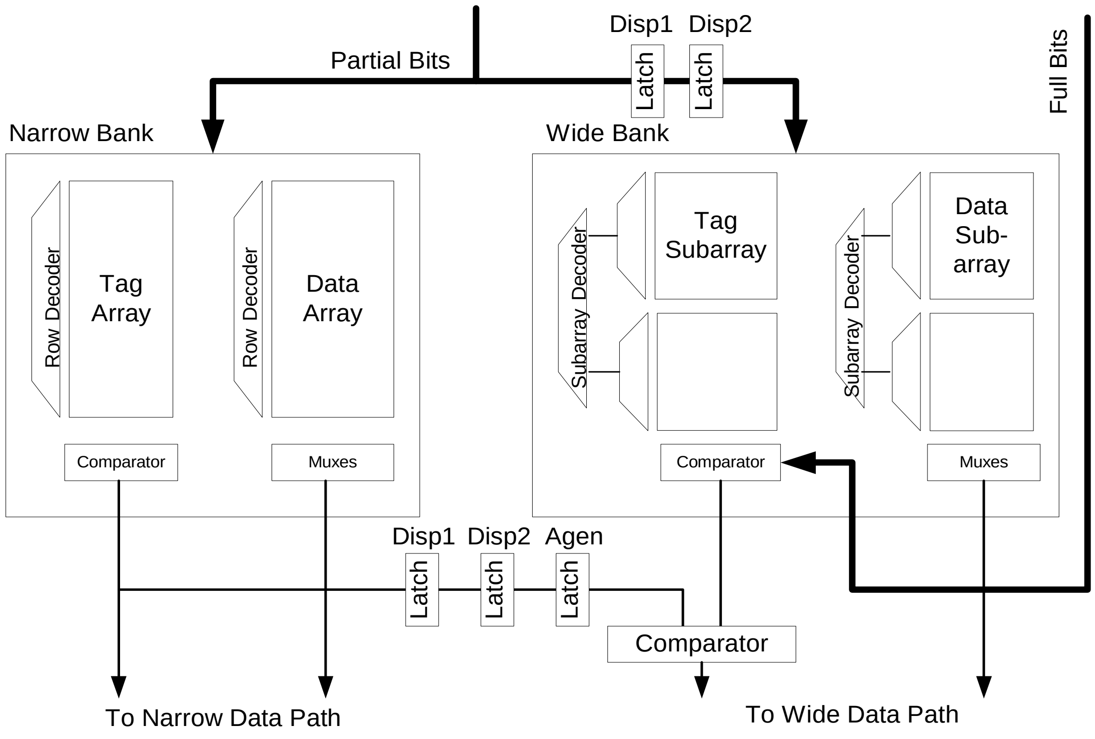


FIGURE 6. Pipelined Cache with Early Partial Bits

nique in our narrow bank due to the latency addition caused by serializing of the subarray and row decoders. Since access time is critical for the narrow bank, any increase should be avoided.

Figure 6 shows the block diagram of our cache. As soon as the partial bits are available, they are sent to the narrow bank to perform the narrow cache access. The hit signal and way select signals are latched to be compared with the result from the wide bank later. Rather than being sent to the wide bank immediately, the partial bits needed to do block select and row indexing are latched until some cycles before the rest of the bits are ready for full tag comparison. The number of cycles between the starting access of the wide bank and the arriving time of the rest of the bits can be tuned based on the amount of work needed before comparison. In the best case, the entire array access can be completed in parallel with computing the upper address bits, leaving only a simple tag comparison for the second cache access pipeline stage. Hence, the cache access latency from the processor's point of view can be reduced from 2-3 cycles into 1 cycle latency.

Since way selection for a set-associative cache is performed as part of the narrow data path, the access to the wide bank can be done using a much simpler and cheaper direct-mapped access. Thus more power can be saved as only a single tag and corresponding data need to be read from the array. This simpler organization has much lower delay than a conventional set-associative cache, and allows designers to use slower, less leaky transistors in the data cache, leading to potentially dramatic reductions in leakage in the level one cache. We leave detailed evaluation of this opportunity to future work.

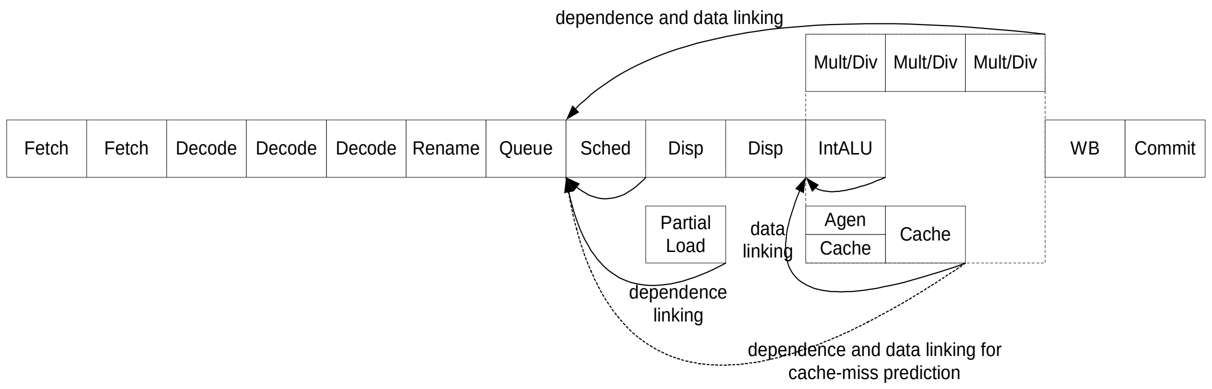


FIGURE 7. Pipelined Diagram with Narrow Scheduler

3.3 Pipeline Integration

Figure 7 shows the pipeline diagram with a narrow scheduler. Simple integer instructions link (or wake up) their dependents in a back-to-back fashion due to the small ALUs placed in the scheduling loop. As can be seen from Figure 7, complex integer operations such as multiplication and division do not wake up their dependents within the scheduling loop. Instead, they wake their dependents up and forward the last sixteen bits of the data after they do full execution in the execution stage. We do not consider floating-point scheduler implementation in this paper, but would expect that a multicycle non-speculative scheduler similar to the approach described for complex integer operations would deliver sufficient floating-point performance.

Load instructions take one or two more cycles to partially access the cache and the store queue before deciding to schedule dependent instructions. If the load experiences a partial tag match in the narrow bank of the cache or store queue, narrow data is forwarded from the appropriate source and dependent instructions are woken up. If the load experiences a partial tag miss, no dependences are linked. Instead, the load will initiate a cache miss after it is done accessing the wide data bank in the execution stage. The load can also experience a partial store queue match with an earlier store that does not yet have its data. In this case, the load is placed back in the scheduler, and dependents are not woken up. In the case where a load experiences a misaligned store alias where forwarding is not possible, no dependents are woken up. Once the alias resolves, a conventional cache access is done and dependents are woken up.

4 Implementation Details

In this section, we describe how we modeled narrow data capture scheduler to get cycle time, area, and power estimates. We also explain the power modeling of the execution pipeline components such as payload RAM, ALUs, cache, and store queue. Lastly, we describe the machine configuration and benchmarks that we used to do performance simulation.

4.1 Scheduler

In order to acquite the information on cycle time, area, and power consumption, we implemented several scheduler designs in Verilog. All design are synthesized using Synopsis Design Compiler and LSI Logic’s gflxp 0.11

Table 2: Cycle Time and Area Comparison

	32 Entries		16 Entries	
	Cycle Time (ns)	Area (mm^2)	Cycle Time (ns)	Area (mm^2)
Full-Data-Capture-Scheduler	2.03	1.26	1.77	0.65
Narrow-Data-Capture Scheduler	1.81	0.44	1.56	0.22
Narrow-Data-Capture Scheduler with Embedded ALUs	1.48	0.60	1.25	0.30
Non-Data-Capture Scheduler	1.35	0.19	1.17	0.10

micron CMOS standard cell library. We experimented with different design constraints for the synthesis such as timing constraint, area constraint, and fan-out constraint. The results shown in this section are the best results obtained from the synthesis tool. It is important to note that the cycle times shown in this section are not directly comparable to the cycle times of custom-designed current generation microprocessors, even if comparable process technology is used. However, we feel that our results are aggressive enough for making useful relative comparisons between the different approaches.

For comparison purposes, we constructed a full data capture scheduler with two 64-bits operands, a narrow data capture scheduler with two 16-bit operands, a narrow data capture scheduler with an ALU embedded in each entry, and a non-data capture scheduler that only stores operand tags in the scheduler entries. For each type of scheduler, we synthesized a 32-entry scheduler and a 16-entry scheduler. Each cycle, all schedulers are capable of issuing four instructions: two simple integer instructions, one memory instruction, and one complex integer instructions. The age of the scheduler entries are maintained using a collapsing logic. We are aware that collapsing logic is not the most efficient way to maintain age priority since, in the worst case, it makes each entry burn power for each cycle. However, due to time constraints and a lack of clear descriptions of alternative age-based selection heuristics in the literature, we opted for collapsing logic in our design.

Table 2 shows the cycle time and the area comparison of different type of scheduler designs. As can be seen from the table, naively capturing sixteen bits of the operands increases the cycle time of non-data capture scheduler by 460 ps in the 32-entry scheduler and 390 ps in the 16-entry scheduler, which is unacceptable. In this case, the increase in the delay is mostly due to the additional time needed to access the narrow ALU. By adding an embedded ALU in each entry, the cycle time increase can be reduced into 130 ps for 32 entries, and 80 ps for 16 entries, since the ALU delay occurs in parallel with the select logic. We feel that this increase in cycle time is reasonable and can be reduced further with careful custom layout.

Table 2 also shows that area increases as more bits are captured in the scheduler. With full operand capture in the scheduler, the area is roughly six times larger than the area of a non-data capture scheduler. When capturing only sixteen bits of operands, the area is roughly doubled; adding narrow ALUs to each entry make it roughly three times larger. Fortunately, with the growing number of transistors on chip, this should not be a problem, though leakage power is always a concern with increased area.

Table 3: Energy Comparison

	32 Entries		16 Entries	
	Collapsing (nJ)	No-Collapsing (nJ)	Collapsing (nJ)	No-Collapsing (nJ)
Full-Data-Capture-Scheduler	1.37	0.78	0.90	0.56
Narrow-Data-Capture Scheduler	0.70	0.45	0.53	0.42
Narrow-Data-Capture Scheduler with Embedded ALUs	0.83	0.46	0.60	0.43
Non-Data-Capture Scheduler	0.49	0.39	0.38	0.34

Table 3 shows the energy dissipated by each schedulers for each access. There are two energy consumption approximations shown for every scheduler. The left column is the energy consumption reported by the synthesis tool on our design that uses collapsing logic. As shown in the table, a narrow data capture scheduler consumes more energy than a non-data capture scheduler due to additional latches needed to store the operands. The scheduler with embedded ALUs consumes slightly more energy compared to the one that has no embedded ALUs since the synthesis tool assumes that all embedded ALUs dissipate power on each cycle.

However, the energy consumption reported here overestimates the real amount of energy consumed by current-generation schedulers due to the fact that collapsing logic is rarely used in real design. The right column is our projection of the energy consumption if a non-collapsing design were used instead. In this projection,we assume that a maximum of four entries in the scheduler will change their contents in one cycle, thus consuming less energy. The remaining entries will keep their contents the same since they are not issued and replaced by new entries. In this scheme, we can see that the additional energy consumed by the operand bits and embedded ALUs becomes much less significant. We will use these projected non-collapsing energy estimates for our total energy consumption estimation in Section 5

4.2 Pipelined Data Cache and Store Queue

We model our pipelined data cache by modifying CACTI 3.0. [13] In order to avoid adding ports to the cache, the cache is separated into a narrow bank and wide bank as explained in Section 3.2. The cache bitslicing technique is employed in the wide bank to save some decoding power. More details on this approach are provided in [10].

Table 4 shows data comparison between our pipelined data cache with bitslicing technique and a conventional data cache. The data shown is for 16KB, 4-way cache with 64B blocks. Two access latencies are shown for the pipelined cache. The first one is the latency to access the narrow bank for partial tag match. That narrow bank access latency added to the scheduler cycle time in Table 2 will be the time needed before scheduling the dependent instructions. Since the narrow cache latency is less than the scheduler cycle time, we believe that only one additional cycle needed after the schedule stage to broadcast narrow data and wake up a load’s dependents.

The access latency of the wide bank for our cache is 1.28ns, slightly larger than the access latency of the conventional data cache that is 1.24ns. This is expected since in our wide bank, subarray decoders are serialized with row decoders to save row decoder power. The access latency spans all the way from subarray decode to the output

Table 4: Pipelined Data Cache vs Conventional Data Cache

	Pipelined Data Cache	Conventional Data Cache
Access Latency - Narrow Bank	0.80ns	N/A
Access Latency - Wide Bank	1.28ns (0.6ns)	1.24ns
Total Energy Consumption	0.37nJ	0.62nJ
Total Area	1.50mm^2	1.21mm^2

mux drivers. However, since we already have the necessary bits to index the cache, the only operations needed after the full bits are available are tag comparison and output mux drive. In Table 4, this number, shown in parentheses, is only 0.6ns.We can use this latency as the cache hit latency in our proposed model, thus reducing the number of cycles needed to access the cache from two cycles into one cycle. Section 5.2 explores the performance benefits that derive from this reduction in cache access latency.

Total energy consumption comparison is also shown in Table 4. Our cache consumes 0.37 nJ per access, significantly lower than the conventional data cache that is 0.62 nJ per access. This energy improvement comes from the fact that we disable unneeded row decoders as in [10]. Since way selection is already performed by the narrow bank, we can save more energy by not doing set-associatives access in the wide-bank. Instead we perform a direct-mapped lookup to the selected way and do tag comparison for the upper address bits.

The area needed for our cache is slightly larger than that of a conventional data cache, but we do not see this as a significant increase relative to total die area. One may argue that this increase in area may cause increase in leakage power. However, the fact that we can start cache access as soon as the partial bits are available from the schedule stage means a slower, less leaky transistors in the cache array can be used. Hence, further power saving from leakage power reduction is possible in our scheme.

Since the store queue is accessed in parallel with the partial cache access, we designed a Verilog model for the store queue with a narrow access port to match the latency of the narrow bank. We found that the latency for a sixteen entries store queue, 0.61 ns, closely matches the latency for the partial cache access.

Table 5: Energy per Access

	Enery per Access (nJ)	
	with Narrow-Data-Capture Scheduler	with Non-Data-Capture Scheduler
Scheduler	0.13	0.10
Payload RAM	0.12	0.16
Store Queue	0.03 / 0.05	0.11
Integer ALU	0.03	0.04
Multiplication/Division Unit	0.85	0.85

Table 6: Area estimation

	Area (mm^2)	
	with Narrow-Data-Capture Scheduler	with Non-Data-Capture Scheduler
Scheduler	0.60	0.19
Payload RAM	0.95	1.26
Store Queue	0.07 / 0.11	0.24
Integer ALU	0.03	0.04
Multiplication/Division Unit	1.02	1.02

4.3 Execution Pipeline Power Modeling

In order to model energy consumption for the execution pipeline, we designed the key units using Verilog. The payload RAM, integer ALU, store queue, and multiplication/division unit were modeled. We assume a machine model as described in Table 7, a 32-entry narrow data capture scheduler with embedded ALUs, matching 32-entry payload RAM, and a 16-entry store queue. We assumed 64-bit operands for the non-data-capture data path, and 16 bits for the narrow data capture data path (the multiplication/division unit is the same for all models). We model two store queues for our narrow-data-capture data path, a small 16-bit fully-associative store queue and a wide 48-bit RAM-based store queue. The premise here is that the narrow store queue provides the index of the aliased store queue entry (if any), thus it is only necessary for the wide store queue to read the full entry at that index and verify the alias condition. Furthermore, that narrow store queue provides a filtering effect that prevents loads that are known to be misses from accessing the wide store queue. This benefit is similar to that achieved by the filtering schemes described in [16] and other related work on store queue scaling.

Table 5 shows the energy per access for each unit. This energy estimation will be used later in Section 5 to estimate total energy consumption for executing some number of instructions. For completeness, we also include the area used for each unit in Table 6.

Table 7: Machine Configurations

	2-Cycle Store-to-Load Forward Latency	3-Cycle Store-to-Load Forward Latency
Out-of-order Execution	4-wide fetch/issue/commit, 64 ROB, 16 LQ, 16 SQ, 32-entry scheduler, 13-stage pipeline, fetch stop at first taken branch in a cycle	
Branch Predictions	Combined bimodal (16k entry) / gshare (16k entry) with a selector (16k), 16-entry RAS, 4-way 1k-entry BTB	
Functional Units	2 integer ALU (1-cycle), 1 integer mult/div (3/20-cycle), 1 general memory ports (1+2), 4 floating-point ALU (2-cycle), 1 floating-point mult/div/sqrt(4/12/24-cycle)	
Memory System (latency)	L1 I-Cache: 64KB, direct-mapped, 64B line size (2-cycle) L1 D-Cache: 16KB, 4-way, 64B line size (2-cycle), virtually tagged and indexed L2 Unified: 2MB, 8-way, 128Bline size (8-cycle) Off-chip memory: 50-cycle latency	
Store-to-Load Forwarding (latency)	Same number of cycle as L1 D-Cache latency (2 cycle)	One cycle longer than L1 D-Cache latency (3 cycle)

Table 8: Benchmark Programs Simulated

Benchmark	input sets	Number of Instructions Simulated (FastForward)	IPC on Base Model (2-cycle / 3-cycle)
bzip2	lgred.graphic	100 M (400 M)	1.35 / 1.35
crafty	crafty.in	100 M (400 M)	1.15 / 1.15
eon	chair.control.cook	100 M (400 M)	1.26 / 1.25
gap	ref.in	100 M (400 M)	0.87 / 0.87
gcc	cccp.i	100 M (400 M)	1.14 / 1.14
gzip	input.compressed	100 M (400 M)	1.01 / 1.00
mcf	lgred.in	100 M (400 M)	0.86 / 0.86
parser	lgred.in	100 M (400 M)	0.71 / 0.71
perlbmk	lgred.markerand	100 M (400 M)	0.77 / 0.76
vortex	lgred.raw	100 M (400 M)	1.34 / 1.34
vprf	ref.net	100 M (400 M)	0.93 / 0.93

5 Simulation Results

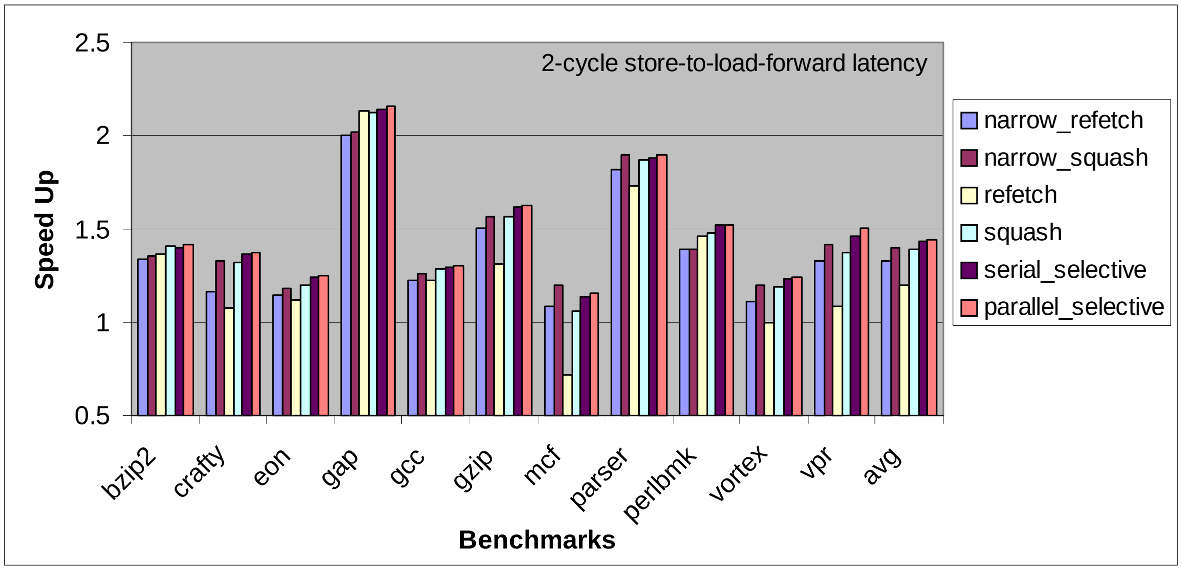
5.1 Simulated Machine Model and Benchmarks

Our execution-driven simulator used in this study is derived from the *Simplescalar / Alpha* 3.0 tool set [14], a suite of functional and timing simulation tools for the Alpha AXP ISA. Specifically, we extended sim-outorder to perform full-speculative scheduling and speculative scheduling with narrow operand knowledge. Various scheduling replay schemes are also modeled in this simulator. In this pipeline, instructions are scheduled in the scheduling stage, assuming instructions have constant execution latency and any latency changes (e.g. cache misses or store aliasing) cause all dependent instructions to be re-scheduled. We modeled a 13-stage out-of-order pipeline similar to POWER4 with 4-instruction machine width. The pipeline structure is illustrated in Figure 7. The detailed configuration of the machine model is shown in Table 7.

The SPEC CINT2000 benchmark suite is used for all results presented in this paper. All benchmarks were compiled with the DEC C and Fortran compilers under the OSF/1 V4.0 operating system using -O4 optimization. Table 8 shows the benchmarks, input sets, the number of instructions committed, and IPC on 4-wide non-speculative machine. The large reduced input sets from [15] were used for all integer benchmarks except for *crafty, eon*, *gap,* and *vpr.* These four benchmarks were simulated with the reference input sets since the reduced inputs are not yet available

5.2 Performance Evaluation

We collected performance results for a non-speculative base machine model with 2-cycle store-load-forwarding latency and 3-cycle store-load-forwarding latency. In both models, we evaluate our proposed narrow capture scheduler scheme with a simple refetch replay scheme and a squashing replay scheme. We compare our scheduler with fully-speculative non-data capture scheduler with four different recovery schemes: refetch replay, squashing replay, serial selective replay, and parall.el selective replay.



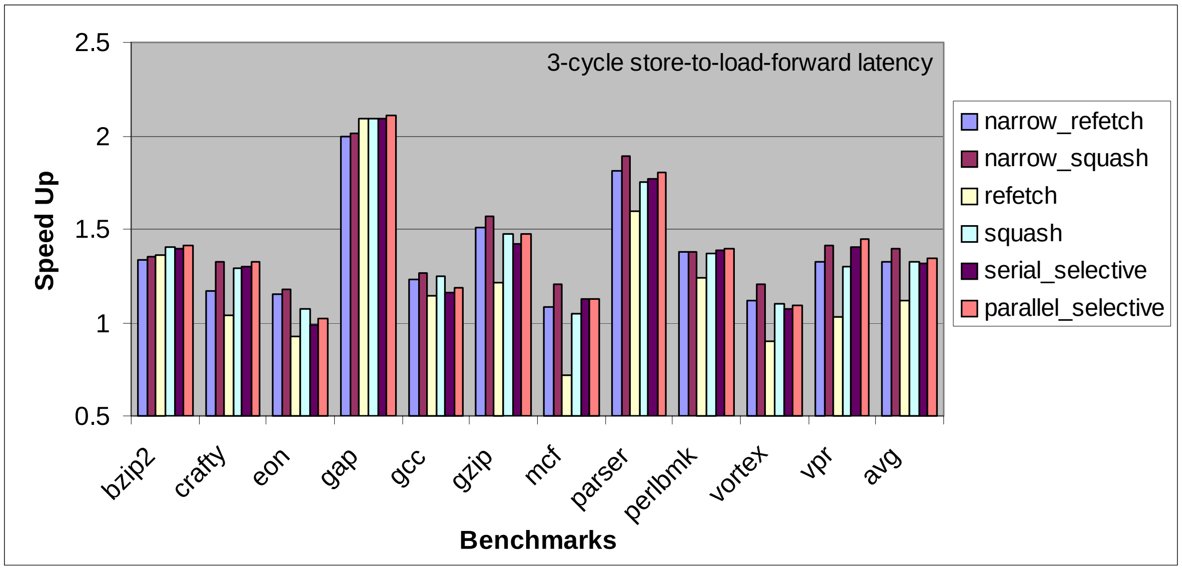
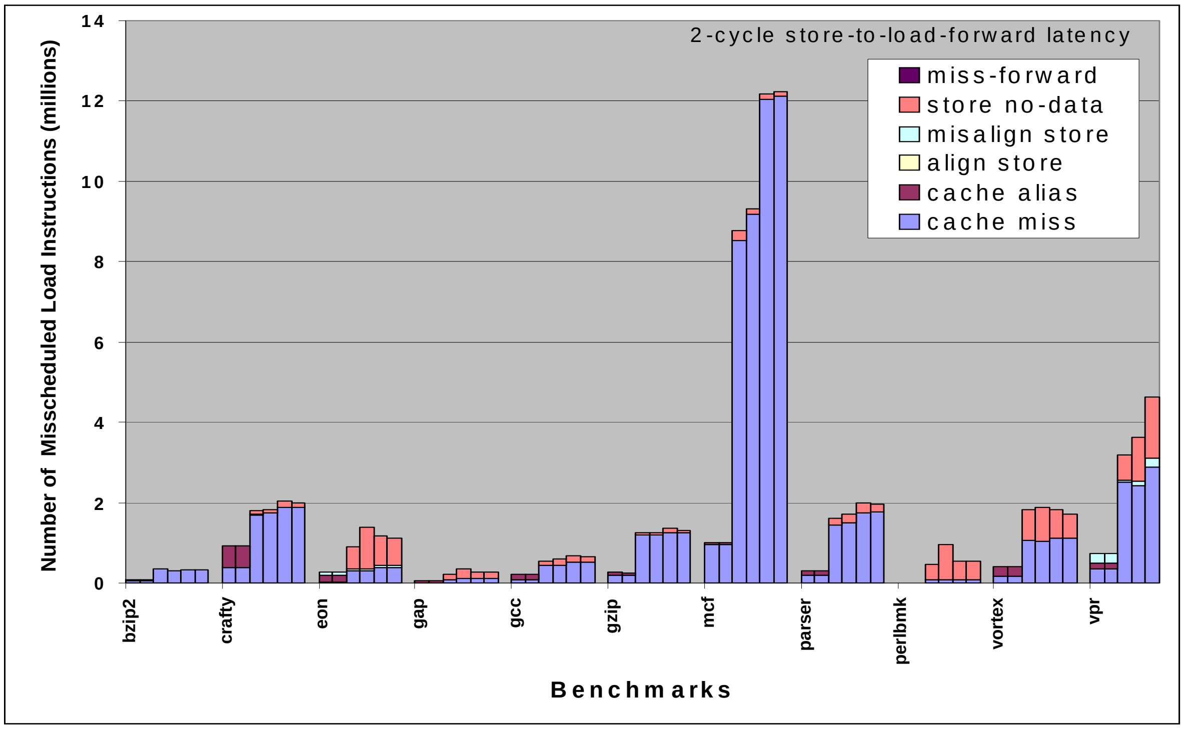


FIGURE 8. Narrow-Capture-Scheduler Speed-Up. Above figures shows performance improvement of different schedulers with different replay schemes over non-speculative base machine.

The speedups in IPC for the narrow-capture-scheduler across SPEC2000int benchmarks are shown in Figure 8. For our 2-cycle load-store-forwarding latency model, in which load-store forwarding does not cause latency misspeculation, narrow capture scheduling with refetch replay scheme performs around 4% worse than speculative-scheduling with squashing replay scheme, 7% worse than speculative scheduling with serial selective replay scheme, and 8% worse than parallel selective replay scheme that is the ideal replay mechanism. Using the squashing replay scheme, the narrow data capture scheduler performs 0.4% better than speculative scheduling with squashing replay, 2.5% worse than speculative scheduling with serial selective replay scheme, and 3.5% worse than speculative scheduling with parallel selective replay scheme..



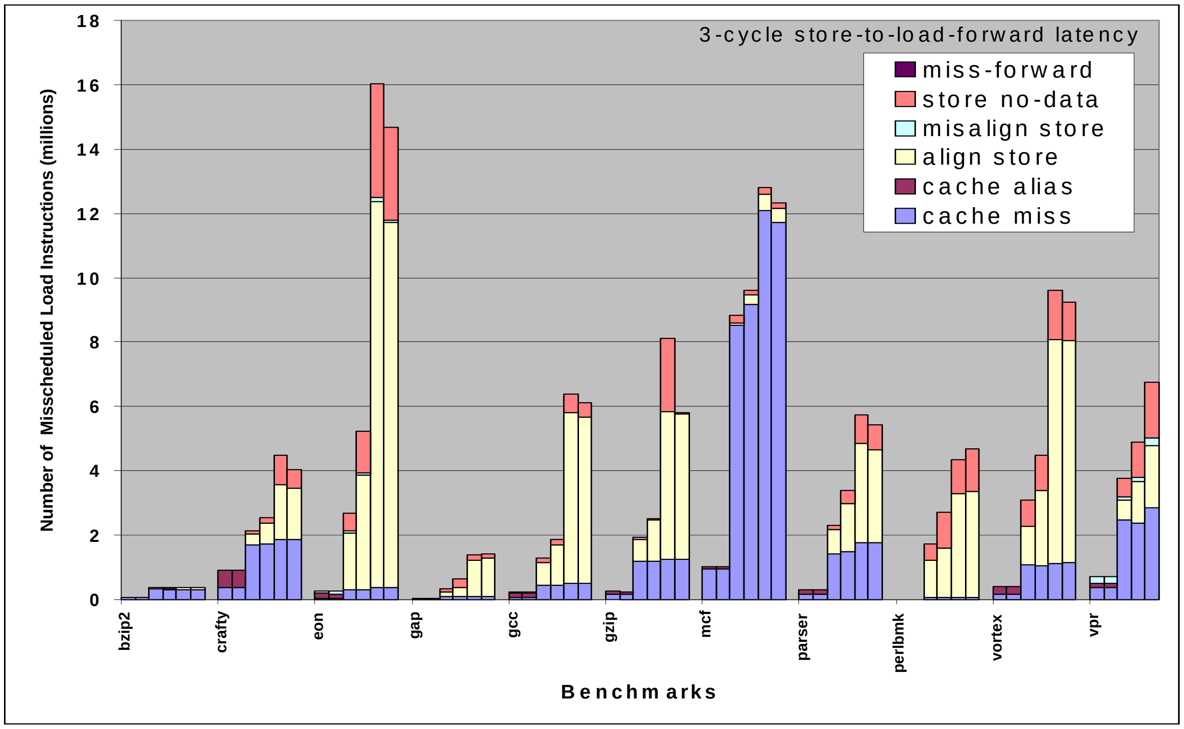


FIGURE 9. Number of Mispredicted Load Instructions. Above figures compares the number of mispredicted load instructions.

For the machine model with 3-cycle store-to-load forward latency, the narrow data capture scheduler performs really well, even better than the best speculative non-data capture scheduler for certain benchmarks. This is due to the fact that store-to-load forwarding takes a cycle longer than load hit latency (this is common in real machines, e.g. the IBM Power4 [4]). Thus, each store alias causes a load misscheduling event. On average, the narrow data capture scheduler with refetch scheme is able to provide performance comparable to a non-data capture scheduler with any replay scheme. With the squashing replay scheme, the narrow data capture scheduler outperforms the non-data capture scheduler with squashing replay scheme by 5.2%, non-data capture scheduler with serial selective replay scheme by 6.0%, and non-data capture scheduler with paralle selective replay scheme by 3.9%.

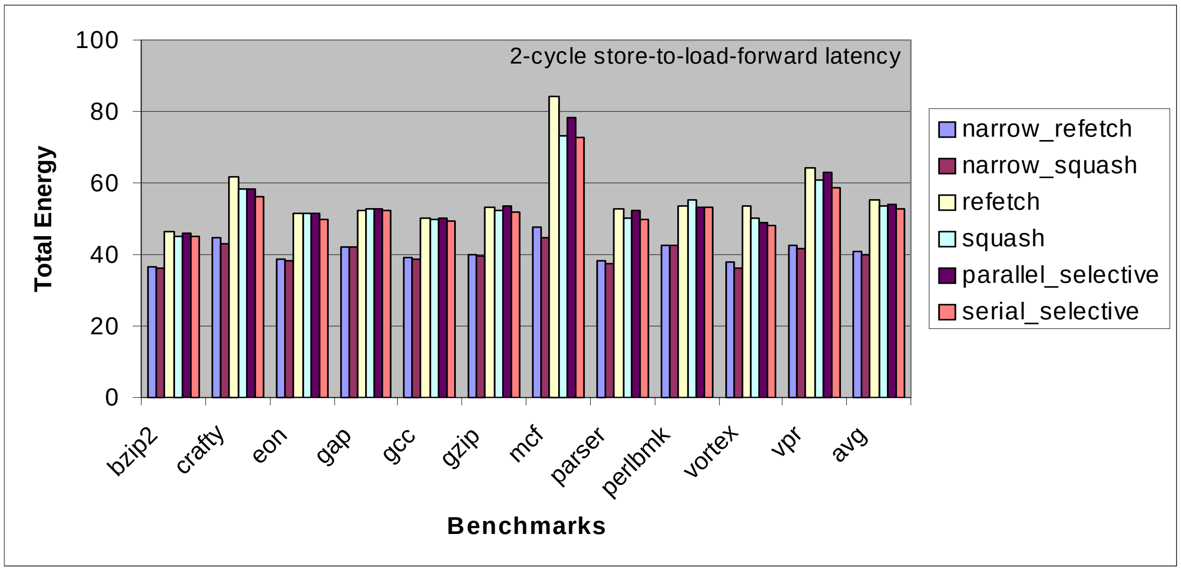
In order to understand our performance number better, we plot the number of misspeculated loads in Figure 9. We categorized the number of misspeculated load instructions into six categories. The first one, cache miss, is a load instruction that is predicted to hit the cache but turns out to miss the cache. The second one, cache alias, is a load instruction that experiences more than one partial tag match in the narrow set-associative tag array, and is predicted to hit one of them while it actually hits another way. This category can only happen in the narrow data capture scheduler. The third category, align store, can only happen when the latency of store-to-load forwarding is longer than cache hit latency. In this case, a load hits an aligned store alias in the store queue and the data needs to be forward from that store. The fourth category is a mis-aligned store alias, where a load hits a misaligned store in the store queue. The fifth category, store no-data, corresponds to loads that hit a store without its store data ready. The sixth category, misforward, occurs due to partial knowledge in the narrow store queue: a load instruction observes a partial match in the narrow store queue, but it turns out that the store is not a real alias to the load.

As can be seen in Figure 9, we can eliminate a significant number of mispredicted loads due to cache misses and almost all mispredicted loads due to store aliasing problems. The results in Figure 9 explain the IPC results in Figure 8. In general, reduction in misspeculated loads means more performance improvement. For example, since there are very few misspeculated load instructions in *bzip* and *gap* there is no performance improvement using narrow data capture scheduler, and sometimes the performance is actually reduced compared to non-data capture machine with the same replay schemes. This is due to the fact that we have to schedule multiplication and division speculatively in the narrow-data-capture scheduler. On the other hand, *mcf* gets a large reduction in misspeculated load instructions and also receives the most performance improvement when narrow data capture is employed.

5.3 Energy Evaluation

In order to estimate the amount of energy consumed by the out-of-order window during program execution, we count the number of times each execution unit, i.e. scheduler, payload RAM, and functional units, are accessed during program execution. These activity numbers are then multiplied by the number of energy dissipated per access by each unit, as shown in Table 5. The energy calculation for the store queue in the narrow data capture scheduler is divided into two parts. The first is the narrow store queue that is accessed by every load instructions. The second one is the wide RAM based store queue, accessed only by load instructions that are predicted to have store aliases in the store queues. Using the narrow store queue, we can filter out the number of access to the wide RAM-based store queue significantly.

As shown in Figure 10, which shows the total energy dissipated by the out-of-order execution window, a significant amount of energy is saved during program execution. The savings are greater for the 3-cycle store-to-load



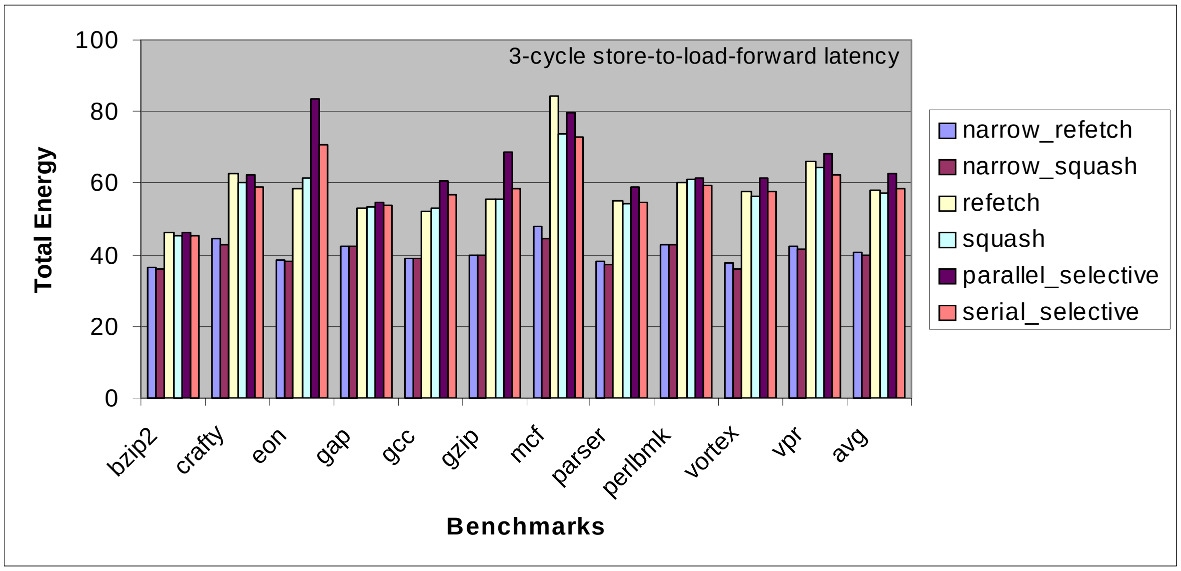
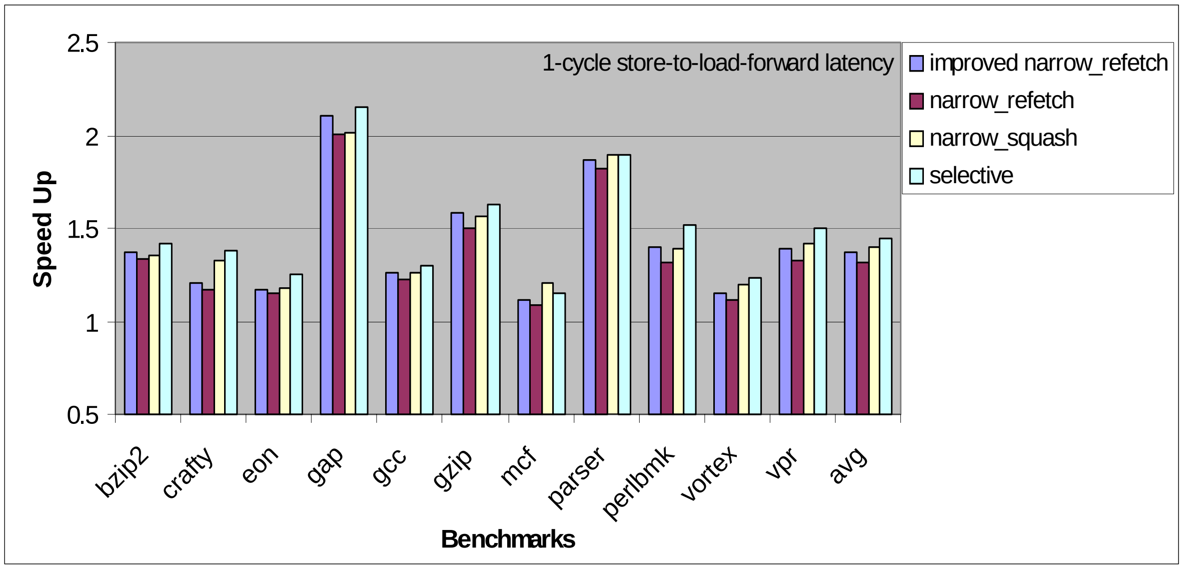


FIGURE 10. Total Energy Dissipation in mJ. Above figures compares energy dissipation by out of order core for during program execution

forwarding case, since there is greater opportunity to avoid misscheduling. On average, the narrow data capture scheduler with squashing replay consumed 25%, 26%, and 24% less energy compared to non data capture scheduler with squashing replay, non data capture scheduler with serial selective replay, and non data capture scheduler with paralle selective replay respectively. These results are conservative since we did not assign any energy penalty to the complex hardware required to implement either selective replay scheme. In the 3-cycle store-to-load forward latency case, the energy saved by the narrow data capture scheduler is 30%, 36%, and 31% with respect to the non-data-capture scheduler with squashing replay, parallel selective replay, and serial selective replay.

Assuming constant cycle time, which is reasonable as seen from our cycle time data in Table 5, it is possible to reduce 25% to 36% of power dissipation by the out-of-order execution window. Since the total power consumption of the out-of-order core is approximately 40%-50% of total power consumed by the chip [5], employing narrow data capture scheduler can save 12%-18% of total chip power consumption, with no significant change in either cycle time



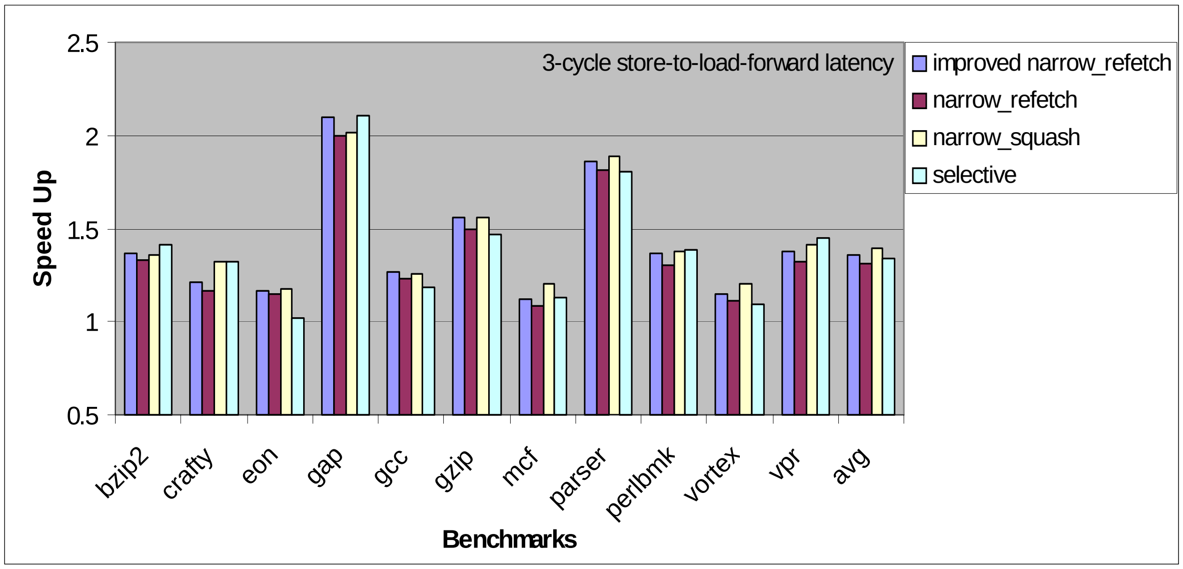


FIGURE 11. Improved Narrow Data Capture Scheduler with Refetch Replay Speed Up.

or IPC.

5.4 Improved Refetch Replay Evaluation

To maintain a fair comparison, the preceding section assumed the same pipeline length and scheduler entry release policies for all options. In this subsection, we evaluate the performance of a narrow data capture scheduler with refetch replay scheme where these policies are improved to match attributes of the scheduler architecture. We exploit the fact that, using refetch replay scheme, there is no need to keep the instructions in the scheduler once they are issued, since a misscheduling event will cause them to be reinserted by the front end. This early issue slot reclaim means that more issue queue occupancy is reduced, and new instructions can enter the window sooner.

Also, since the partial bits enable cache access to start early in the pipeline, the results can be returned soon after the rest of the address bits are computed by the load address generation. In this improved refetch replay configuration, we assume one cycle cache access latency rather than two cycles latency. As shown in Table 4, this is a reasonable assumption.

Table 11 shows the performance gained by our optimal narrow data capture scheduler with refetch replay scheme. For the first graph, we assume that the store to load forward takes the same number of cycles as cache hit latency, which is one cycle latency. For the second graph we assume that store to load forwarding remains 3 cycles while cache hit latency becomes one cycle. As shown, our improved scheme achieves comparable performance with narrow data capture with squashing replay. In the 3 cycles store-to-load forward latency, our improved scheme performs slightly better than non data capture scheduler with selective recovery.

6 Conclusions and Future Work

This paper proposes a technique to reduce the number of misscheduling replays in microprocessors using partial operand knowledge. Specifically, sixteen bits of each operand are captured in the data scheduler and used to do early load disambiguation and partial tag matching. On average, we are able to reduce the number of misspeculated loads by 75% to 80% when store-to-load forwarding takes the same number of cycles as a cache hit and 80% to 90% when store-to-load forwarding takes longer than a cache hit.

Using this technique, we can also employ less complex recovery mechanism without losing much performance. With a narrow data capture scheduler, a simple refetch replay scheme performs comparably with a complex squashing replay scheme on a non-data-capture scheduler, while a squashing replay scheme performs comparably with parallel selective replay on a non-data-capture scheduler. When store-to-load forwarding takes longer than a data cache hit, squashing replay with our scheduler actually outperforms a serial selective recovery scheme.

Finally, since fewer instructions are being replayed, and load accesses to the cache and store queue consume less energy, we save a significant amount of energy during program execution. We can save approximately 25% to 36% of total out-of-order window energy across different recovery schemes. Since the out-of-order execution core consumes 40%-50% of total chip power, our technique should save 12%-18% of total dynamic chip power.

We believe that more opportunities can be exploited by our narrow capture scheduler. One possibility is to create a very wide issue window by combining a narrow data-capture scheduler with narrow operand values. Since we have as many embedded ALUs as the number of entries in the scheduler, all of the integer calculations with reduced operand significance could be completed in the schedule stage. The embedded ALUs also provide hardware redundancy, which could be exploited further for transient or permanent fault detection.

As mentioned earlier, since the partial bits needed to do cache access are available very early in the pipeline, we do not need a very fast cache design that employs fast, leaky transistors. Instead we can use slow transistors, start the access early, and reduce leakage power in the cache array. We plan to study such opportunities for leakage power reduction in future work.

References

[1] G. Hinton et al., The Microarchitecture of the Pentium 4 processor, *Intel Technology Journal* Q1, 2001.

[2] Compaq Computer Corporation, Alpha 21264 microprocessor hardware reference manual, July 1999.

[3] I. Kim and M.H.Lipasti, Understanding Scheduling Replay Schemes, In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, February 2004.

[4] D.C.Bossen, J.M.Tendler, K.Reick, Power 4 System Design for High Reliability, In *IEEE Micro*I, Vol. 22, pp.16-24, March-April 2002.

[5] D. Brooks, V. Tiwari, M. Martonosi, Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, In *Proceedings of the 27th International Symposium of Computer Architecture*, 2000.

[6] B.Mestan and M.H.Lipasti, Exploiting Partial Operand Knowledge, In *Proceedings of the 3rd International Conference on Parallel Processing*, 2003.

[7] D.Brooks and M.Martonosi, Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, In *Proceedings of the 5th International Symposium on High-Performance Compute Architecture*, January 1999.

[8] L.Liu, Cache Designs with Partial Address Matching, In *Proceedings if the 27th Annual International Symposium on Computer Architecture*, December 1994.

[9] J.P.Shen and M.H.Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, beta edition, pp. 179-181, McGraw-Hill, 2002.

[10] E.Gunadi and M.H.Lipasti, Cache Pipelining with Partial Operand Knowledge, In *Proceedings of the Workshop on Complexity-Effective Design*, in conjunction with ISCA, June 2004.

[11] S.J.E. Wilton and N.P. Jouppi, CACTI: An Enhanced Cache Access and Cycle Time Model, In *IEEE Journal of Solid-State Circuits*, May 1996.

[12] G. Reinman and N.P.Jouppi, CACTI 2.0: An Integrated Cache Timing and Power Model.

[13] P.Shivakumar and N.P.Jouppi, CACTI 3.0: An Integrated Cache Timing, Power, and Area Mode.

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

[15] A.KleinOsowski, J.Flynn, N.Meares, D.J.Lilja, Adapting the SPEC2000 benchmarks suite for simulation-based computer architecture research, *Workshop on Workload Characterization in Internationl Conference on Computer Design*, 2000.

[16] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, Stephen W. Keckler, Scalable Hardware Memory Disambiguation for High ILP Processors, Proceedings of MICRO-2003, November 2003.