Journal of Instruction-Level Parallelism 9 (2007) 1-23 Submitted 10/06; published 4/07

Narrow Width Dynamic Scheduling

Erika Gunadi EGUNADI@ECE.WISC.EDU
Mikko H. Lipasti MIKKO@ECE.WISC.EDU

Department of Electrical and Computer Engineering
1415 Engineering Hall
University of Wisconsin
Madison, WI 53706

Abstract

To satisfy the demand for higher performance, modern processors are designed with a high degree of speculation. While speculation enhances performance, it burns power unnecessarily. The cache, store queue, and load queue are accessed associatively before a matching entry is determined. A significant amount of power is wasted to search entries that are not picked. Modern processors speculatively schedule instructions before operand values are computed, since cycle-time demands preclude inclusion of a full ALU and bypass network delay in the instruction scheduling loop. Hence, the latency of load instructions must be predicted since it cannot be determined within the scheduling pipeline. Whenever mispredictions occur due to an unanticipated cache miss, a significant amount of power is wasted by incorrectly issued dependent instructions.

This paper exploits the prevalence of narrow operand values by placing fast, narrow ALUs, cache, and datapath within the scheduling loop. The results of this narrow datapath are used to avoid unnecessary activity in the rest of the execution core by creating opportunities to use different energy reduction techniques. A novel approach for transforming the data cache, store queue, and load queue from associative (or set-associative) to direct mapped saves a significant amount of energy. Additionally, virtually all load latency mispredictions can be accurately anticipated with this narrow datapath, and very little energy is wasted on executing incorrectly scheduled instructions. Our narrow datapath design, coupled with a novel partitioned store queue and pipelined data cache, can achieve cycle time comparable to those of conventional approaches, while dramatically reducing misspeculation. This technique saves approximately 27% of the dynamic energy of the out-of-order core, which translates into roughly 11% of total processor dynamic energy, without any loss of performance for integer benchmarks. Finally, a less-complex flush-based recovery scheme is shown to suffice for high performance due to the rarity of load misscheduling.

1 . Introduction and Motivation

Modern processors rely heavily on aggressive memory hierarchies to reduce average memory reference latency and supply adequate memory bandwidth. As a necessary consequence, memory references are satisfied with variable latency, since not all requests can be serviced from the fixed-latency primary data cache, due to its limited capacity. Conventional out-of-order processors are capable of tolerating variable latency by finding other useful instructions to execute in the shadow of primary cache misses. However, as the relative memory latency and variability in latency have increased for each successive processor generation, the ability to efficiently tolerate this variability has actually diminished, since aggressive cycle-time constraints have effectively precluded dynamic scheduling logic that can react instantaneously to hit/miss status signals from the memory hierarchy. As a result, the dynamic scheduling logic in today's processors assumes that all references will hit in the primary data cache, and creates a speculative execution schedule based on that assumption. Whenever the assumption proves to be incorrect (whenever the primary cache

misses), a new schedule must be created and in-flight instructions must be flushed out of the pipeline.

In order to minimize the occurrence of these pipeline flushes, primary caches are designed to maximize fixed-latency hits; this leads to a set-associative organization where multiple tags (typically four) are checked in parallel to find a matching entry. Similarly, the store queue, which may contain the most recent value for a memory location, is checked in parallel, using a fully-associative lookup, even though it actually supplies that value relatively rarely (it mostly misses). Whenever it does hit, the delay path to satisfy the load must be balanced so as not to perturb the dynamic scheduler's fixed-latency assumption. The load queue is similarly checked to detect loads that were ordered incorrectly, both with respect to earlier stores from the same processor as well as remote writes from other processors on the system.

All of these factors—the need for speculative scheduling, set-associative lookup in the primary data cache, and fully-associative lookup in the load and store queue—collude to consume a significant portion of the processor's power budget to conduct inherently unnecessary activity. If only the processor could identify hits and misses ahead of time, it would not need to schedule dependent instructions speculatively, and could avoid useless speculation. If only the data cache knew which way of the cache was going to hit, it could avoid the overhead of parallel lookups in the other *n-1* ways. Finally, if only the store queue knew which entry would conflict with an issued reference, it could access that entry directly, avoiding an expensive and power-hungry associative lookup.

This paper proposes a novel significance-based organization for the processor's dynamic scheduler and data path that achieves all of these goals. A well-known program attribute—that most integer computations have relatively few significant bits—inspires a design that incorporates a narrow data path and nonspeculative dynamic scheduler into an otherwise conventional out-of-order processor. Our results show that the narrow scheduler can be fully reactive and nonspeculative, since its area and logic delays are substantially reduced relative to an equivalent full-data-width design. The early availability of narrow operand results from the narrow-width scheduler enables the narrow datapath to identify useless speculation and obviates any need for speculative scheduling, set-associative cache lookup, or fully-associative store queue lookup in the wide portion of the machine. Instead, these operations are replaced with a fully nonspeculative data path, primary cache, and store queue, that are governed by the outcomes created by the power-efficient, nonspeculative narrow core that can *a priori*, with virtually 100% certainty, identify cache misses; cache hits and which cache way will supply them; as well as store queue hits. This information is used to control the conventional wide core, eliminating any need for speculation and converting set-associative and fully-associative lookups in to power-efficient direct-mapped accesses.

It is well known that full operand bits are not necessary to do load-store disambiguation or to determine cache misses [1][2]. 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 large percentage of cache hits (tag matches) and misses can be determined by using only the first 15-20 bits of the address tags[1]. In fact, the Pentium 4 [3] processor addresses its level-one data cache using only 16 bits of the virtual address. Hence, we show that capturing some number of bits of the operands in the scheduler can effectively determine load latency in virtually all cases.

While earlier work has pointed out that partial operands are sufficient for correctly identifying cache hits and misses and for resolving load/store aliases, this paper is the first to describe an ele-

1. We show the worst-case (mcf) and a typical case (bzip2); other benchmarks are substantially similar.

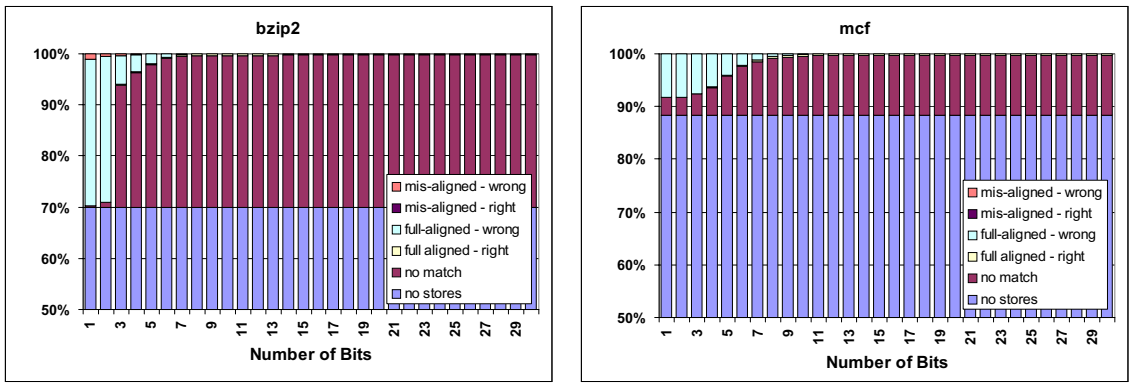

Figure 1: Percentage of Load-Store Disambiguation per Number of Bits. No stores refers to no stores in front of the load. No match refers to no store match in lsq. Full-aligned refers that the partial bits predict a prior fully-aligned store. Mis-aligned refers to partial bits predict a non-fully-aligned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

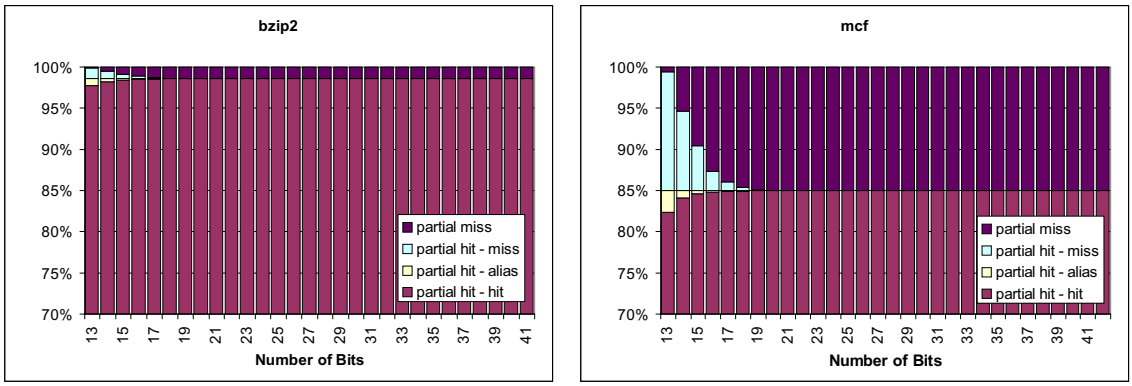

Figure 2: Percentage of Partial Tag Match per Number of Bits. Partial hit - hit means that partial bits predict a load to be and the prediction is right. Partial hit - alias and partial hit - miss refers to a cache hit prediction that is wrong due to associative aliasing and cache miss. Partial miss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

gant and attractive approach for restructuring the entire execution pipeline to exploit this program property. Namely, we propose dynamic scheduling logic that uses partial operand knowledge (the least significant bits) to create a nonspeculative execution schedule that is fully reactive to cache hit/miss results from our narrow data cache and store queue.

Since the wakeup-select scheduling loop is usually one of the critical paths in a microprocessor, we must design a narrow data-capture scheduler that is no slower than an equivalent non data-capture scheduler. To establish the validity of our proposal, and to quantitatively evaluate performance and power consumption (dynamic and static), we created a detailed register-transfer-level implementation of all key components, synthesized using a modern standard cell design flow. Our implementation compares favorably in terms of cycle time to industrial standard cell-based designs (our target frequency is 50% higher than [24], an optimized industrial design in a slightly older technology). Detailed analysis of dynamic and static power shows that we can dramatically reduce power consumption with our sliced out-of-order core. We also show that the complexity of the scheduler can be reduced by using a less aggressive recovery mechanism while sacrificing lit-

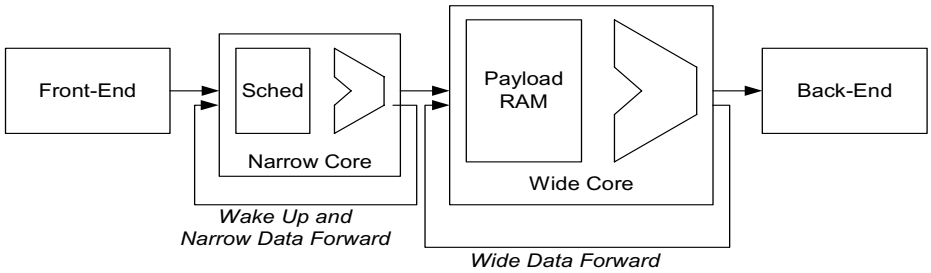

Figure 3: Block Diagram for Dynamic Scheduling with Partial Operand Knowledge

tle performance.

The rest of the paper is structured as follows. Section 2 describes the details of narrow data-capture scheduler and pipeline integration. Section 3 explains the modeling of the scheduler, execution pipeline power consumption, and microarchitectural model. Section 4 provides detail performance and power evaluation of our narrow data-capture scheduler. Section 5 describes background and related work and Section 6 concludes the paper.

2 . Narrow Width Dynamic Scheduling

We propose to use partial operands captured in the narrow data-capture scheduler to reduce useless speculative activity. Partial tag match [1] and partial load-store disambiguation [2] are used to determine load latency. Partial tag match uses partial tag bits to generate an early hit/miss signal and confirm it after the full tag bits are compared. Partial load-store disambiguation uses a similar idea to determine if there is a matching store alias entry in the store queue. Additionally, partial operands knowledge is also used to eliminate unnecessary associative accesses on load queue, store queue, and cache. As shown in Figure 3, the out of order core is sliced into a narrow core and a wide core. The narrow core, which consists of a narrow data-capture scheduler, narrow ALUs, narrow load queue, narrow store queue, and narrow cache bank, uses partial operand knowledge to control the rest of the execution.

Sixteen bits of the operands are stored in the scheduler. Narrow ALUs compute the partial data to be forwarded to the dependent instructions as well as partial addresses to be sent to the narrow load queue, narrow store queue, and narrow cache bank. In the case of a store alias, partial data from the store queue is forwarded to dependent instructions. Otherwise, partial data from the narrow cache bank is used. Our scheduler captures sixteen bits as this subset of operands is enough to reliably determine store alias and cache hit/miss while being small enough to achieve fast cycle time, as we will see in Section 3.

The early availability of partial operands also enables us to employ a novel associative - direct mapped transformation approach for cache, load queue, and store queue. The results of the narrow load store queues and narrow cache bank accesses are used to access the corresponding wide queues and wide cache in power-effective direct-mapped fashion. Payload RAM and execution units are also clock gated when both operands are narrow. All these techniques are incorporated in a detailed timing and power model that accounts for both dynamic and static power and show that significant savings are possible with no detrimental impact on performance.

The rest of Section 2 will be presented as follows. Section 2.1 gives a brief overview of basic scheduler design. Since achieving a cycle time no longer than a conventional non data-capture scheduler is one of our biggest challenges, we discuss the details of our design in Section 2.2. Section 2.3 discusses our pipelined cache design. Load queue and store queue details are presented in Section 2.4. Finally, Section 2.5 describes the whole pipeline.

2.1. Scheduling Basics

Based on where operands are stored, schedulers are divided into data-capture schedulers and non data-capture schedulers [4]. 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. A data-capture scheduler copies operands from the register file when dispatching instructions. For not-ready operands, tags are copied and used later to latch operands when they are forwarded by the functional units. In effect, result forwarding and instructions wake up are combined in a single physical structure. Section 2.2 describes how to build a data-capture scheduler that incorporates only a slice of the datapath, and is able to achieve a competitive cycle time. In contrast, no operands are copied into a non data-capture scheduler; rather, only tags for operands are loaded into the window. The scheduler still performs tag matching to wake up ready instructions. 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.

While non data-capture schedulers are able to achieve faster cycle time due to the decoupling of scheduling from register file read and bypass, the additional pipeline stages needed for register file access and ALU execution prevent the scheduling of dependent instructions in a back-to-back fashion. One natural solution is a speculative scheduler, in which dependent instructions are issued before actual execution occurs. The scheduler assumes that instructions have predetermined fixed latencies and schedules the dependences based on those latencies. In the case of misscheduling due to latency misprediction, a recovery mechanism is invoked to replay misscheduled instructions.

2.2. Narrow Data-Capture Scheduler

As mentioned earlier, our narrow data-capture scheduler stores the least significant sixteen bits of the input operands. To compute partial data to be forwarded to the dependent instructions, narrow ALUs need to be added to the scheduling loop since both scheduling and execution must occur in the same cycle to avoid speculative scheduling. As a result, the cycle time of our scheduler will be naturally longer than that of a non data-capture scheduler, which only needs to perform wakeup and select. As it is well known that scheduler loop is one of critical paths in determining microprocessor cycle time, an increase in the scheduler cycle time is highly undesireable. However, with careful design outlined below, it is possible to achieve cycle time that is as fast as that of a non data-capture scheduler.

A naive implementation of a scheduler with a narrow data path is shown in Figure 4(a). 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 are added to see whether a load has a store alias or whether it misses the cache. Not shown in the picture is also a narrow load queue for a store to check if there are consistency-model violations due to reordered loads. In addition to tag broadcast buses, narrow data broadcast buses are added to forward the data back to the scheduler. Floating point operations and complex integer operations such as division and multiplication have to schedule their dependent instructions non-speculatively since it is difficult to break such functional units into narrow and wide units. The same is applied to right shift operation. Though not as complex, supporting narrow right shift operations would require an additional bit to be captured by the scheduler.

In contrast to a non data-capture scheduler, our narrow data-capture scheduler has two possible critical paths. As shown in Figure 4(a), the first path, shown with a solid line, is the tag broadcast loop. The second path, shown with a dotted line, is the data forwarding loop. Based on Figure 4(a), the likely critical paths are:

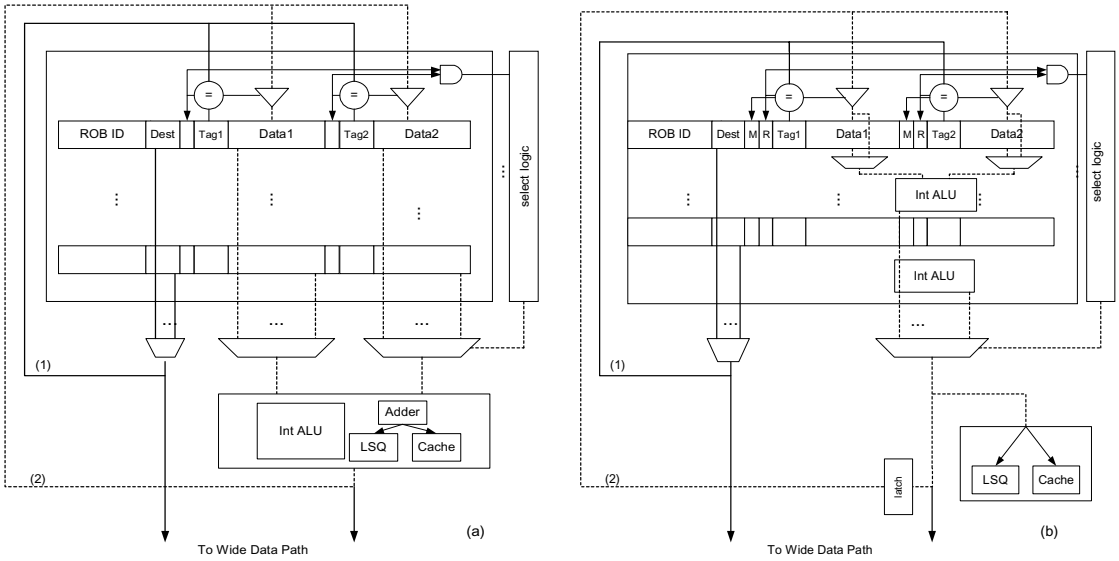

Figure 4: (a) Scheduler with Narrow Data-Path, (b) Scheduler with Embedded Narrow ALUs

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)

Since both the select logic and the narrow ALU contribute substantial delay, the second delay path is actually the critical path in our narrow data-capture scheduler. By trading additional area for reduced delay, the narrow ALU latency can be hidden for integer ALU instructions by embedding narrow ALUs in every scheduler entry. Since each entry now has its own narrow ALU, narrow integer ALU execution does not have to wait for select logic results and can be done in parallel with the select operation. Now the second delay path will be *max(select, narrow ALU)+mux+data broadcast+data write.* Assuming that the select operation takes longer than the ALU computation, the narrow ALU delay will be completely hidden. However, this new delay is still not as fast as the first delay path since the data write operation depends on the result of the tag comparison from the first delay path. As generating and writing the ready signal is faster than selecting and writing data, the second delay path will still be the critical path.

The second delay can be further reduced by deferring the selecting and writing process to the next cycle by relocating the pipeline latch. The path will now be *max(select, data broadcast+mux+narrow ALU)+mux+latch setup* as shown in Figure 4(b). Assuming that the latch setup takes less time than tag broadcast, compare, and ready signal write, we can expect that now the delay of our narrow data-capture scheduler will be approximately the same as a non data-capture scheduler.

The improvement in the scheduler delay above will not noticeably improve the latency of load instructions since the store queue 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 the load latency has elapsed. Since partial cache access and partial

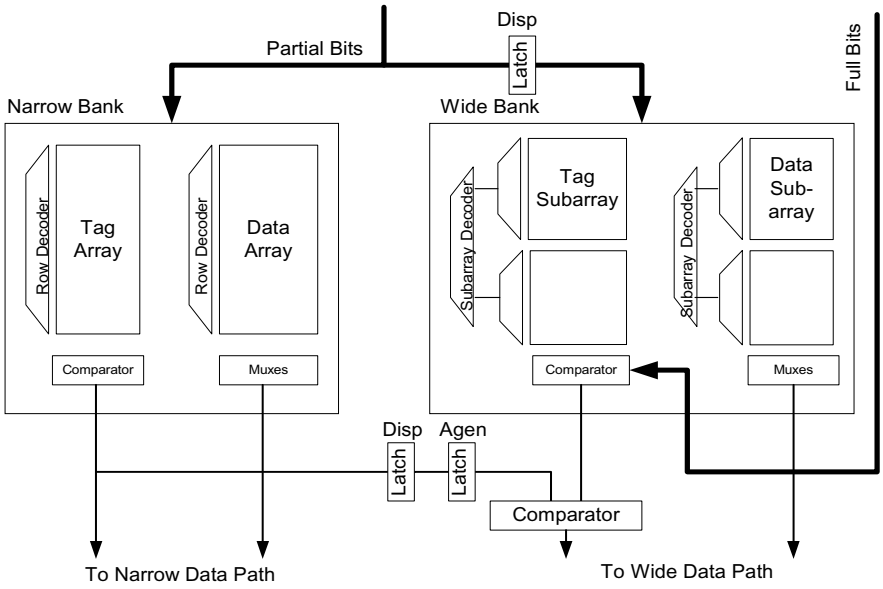

Figure 5: Pipelined Cache with Early Partial Bits

store queue access need less time than a full cache access and a full store queue access, these actions can easily be completed in time to wake up load-dependent instructions despite the additional select delay.

2.3. Pipelined Data Cache

To enable cache access using partial data, our scheme utilizes 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 [16] to activate only a single row decoder and subarray in the wide bank, conserving decoder and subarray access power. This technique is not employed in our narrow bank due to the additional latency caused by serializing the subarray and row decoders. Since access time is likely to be critical for the narrow bank, any such increase should be avoided. To make the wide bank access simpler, narrow tag aliasing in the narrow tag array is not allowed.

Figure 5 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 power-efficient direct-mapped access. Thus, more energy can be saved as only a single tag and corresponding data need to be read from the array.

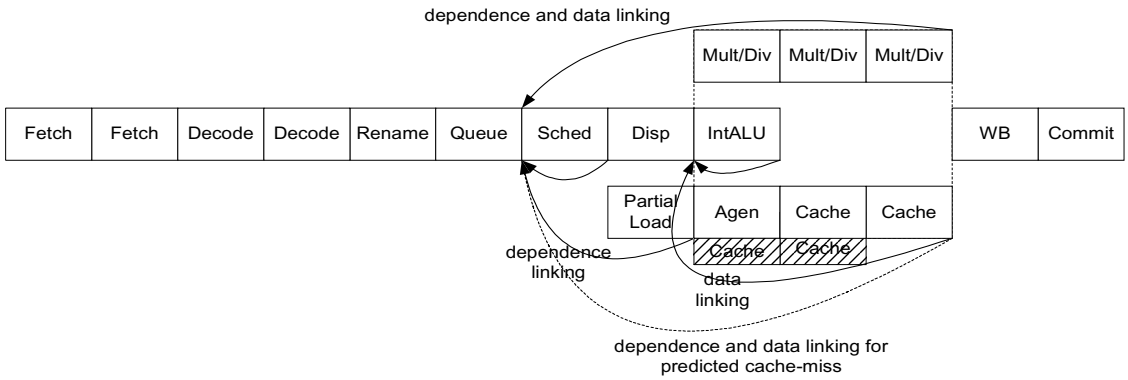

Figure 6: Pipelined Diagram with Narrow Scheduler. Striped area represents optimized model

This simpler organization has much lower delay than a conventional set-associative cache and allows designers to use leakage reduction techniques or slower, less leaky transistors in the data cache, leading to potentially dramatic reductions in leakage in the level one cache.

2.4. Load Queue and Store Queue

Load queue and store queue are divided into narrow and wide queues. Narrow queues are located in the narrow core and accessed using partial address computed by narrow ALUs. Sixteen bits of addresses are stored as tags and sixteen bits of data are also captured to facilitate data forwarding.

The narrow queues are accessed in the conventional set-associative fashion, meaning that in the case of several matches, the youngest among older stores is chosen. The index of the matching entry is then used to access store queues and load queues later in the wide queues, avoiding an expensive fully-associative access. In the rare case when the wide-tag comparison proves to be a false hit, the pipeline is flushed and the load is replayed non-speculatively. In some cases where the narrow queues conclude that there is no aliases, the wide queues can be gated as accesses are not necessary.

2.5. Pipeline Integration

Figure 6 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 shown in Figure 6, 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. 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. A more detailed out-of-order portion of the pipeline is shown in Figure 7. In parallel with reading upper portion of operands from payload RAM, load instructions access narrow queues and narrow cache bank. For fair comparison, access to the side cache bank is done after upper address generation is completed. However, wide bank accesses can start early together with generating the upper portion of the address as explained in Section 2.3. Full address will then be used to do full tag comparison and to access wide load and store queues. This optimized pipeline is shown in striped area. In this case, exposed cache latency becomes one cycle rather than two cycle. Both the naive and the optimized pipeline model is quantified in Section 4.3

If the load experiences a partial tag match in the narrow bank of the cache or store queue, nar-

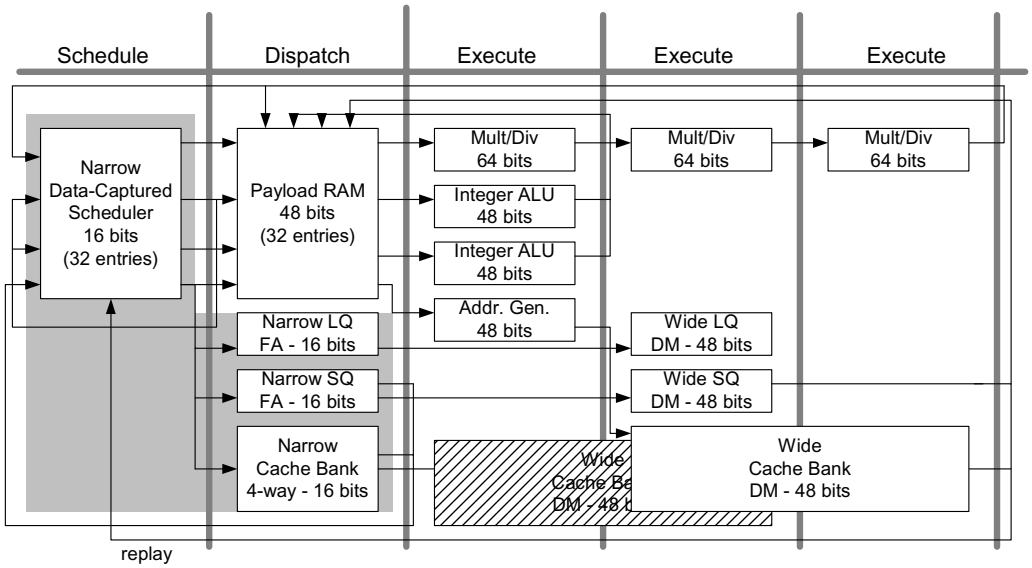

Figure 7: OoO core block diagram. Shaded area is narrow core. Striped area represents optimized model

row 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 initiates 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 the dependents are woken up.

Not all load instructions can have partial data to be forwarded. Since our narrow data-path only operates on the least sixteen significant bits, load-byte instructions that access the third byte or above and load-word instructions that access the upper half of the word will not find partial data to be forwarded. In this case, the load is scheduled non-speculatively and no dependences are linked. We have found that this case is rare enough that it does not significantly affect performance.

We also clock-gate the upper portion of the payload RAM and ALU when both operands are narrow by adding a narrow-check circuitry between the architected register file and the issue queue. When an operand read from the register file has its higher 48-bits as all zeroes or ones, a flag is set in the issue queue entry. If an issued instruction has this flag set for both operands, access to the wide payload RAM and ALU is disabled.

To summarize, our narrow data-captured scheduler provides partial operands early in schedule stage to create opportunity for reducing energy. In this scheme the energy saving comes from 1) reducing issued instruction due to more accurate scheduling decision, 2) transforming the wide portion of the load queue, store queue, and cache into cheap direct-mapped structures, 3) disabling wide load and store queues in no-aliasing cases, 4) disabling payload RAM and ALUs in narrow operation cases. It is important to note that even though we employ some previously-proposed techniques such as partial tag match [1], partial load-store disambiguation [2], and wide data-path clock gating [14], we are the first to describe a simple and elegant overall framework for implementing these schemes that relies entirely on our novel narrow data-capture scheduler design. The

reader should realize that without our novel scheduler, the existing techniques would be impractical or only marginally useful. Applying partial tag match and partial load store disambiguation naively only reduces misscheduling replay penalty by 1-2 cycles, which means many issued instructions still have to be replayed. Similarly, clock gating the wide functional units saves only a marginal amount of dynamic power as shown in Section 4.2.

2.6. Design Complexity

Though our proposal may seem complicated, we have found that our narrow data-capture scheduler design does not add much complexity to the existing out-of-order core. Our narrow datapath is technically a subset of a regular speculative execution core. As can be seen from Figure 7, the main modifications are in the scheduler itself, load store queues, and cache. As explained in Section 2.2 and Section 2.4, the changes to scheduler and queues are straightforward. As cache banking has been widely used to reduce number of ports, our narrow-wide cache banking could be done in similar way. Additional features to the narrow datapath are load-byte, load-word checking, and non-speculative scheduling for complex integer instructions; none of these are particularly challenging. Furthermore, since there is no scheduling done in the wide datapath, it is far less complex than a conventional execution core. Finally, the fact that a single graduate student, working for less than 12 months, has successfully implemented all of the key portions of the proposed datapath at the register-transfer level, lends credence to our claim that the complexity is quite manageable.

3 . Implementation Details

This section describes how we modeled narrow data-capture scheduler to arrive at cycle time, area, and energy estimates. We also explain the modeling of the execution pipeline components such as payload RAM, ALUs, cache, store queue, and load queue.

3.1. Scheduler

To acquire the information on cycle time, area, and power consumption, several scheduler designs are implemented in Verilog and synthesized using Synopsis Design Compiler and LSI Logic's gflxp 0.11 micron CMOS standard cell library. Due to tool inavailability, designs have not been placed and routed. However, the synthesis tool does include an estimate of the wire delay using fanout-based wire load models as the tool has not yet known the exact wire lengths and the capacitive loads. According to [25], synthesis reasonably estimates the wire load of the multitude of shorter wires while underestimating the longer nets. Thus, in addition to synthesis result, a wire delay estimation using distributed RC model is done for data and tag broadcast critical path.

We experimented with different design constraints, such as timing, area, and fan-out for synthesis. The results shown in this section are the best results obtained from the synthesis tool. It is important to note that cycle times shown in this section are not directly comparable to cycle times of custom-designed current generation microprocessors, even if comparable process technology is used. However, our implementation compares favorably in terms of cycle time to industrial standard-cell designs (our target frequency is 50% higher than [24], an optimized industrial 8-pipe-stage design in a slightly older technology). Hence, our results are aggressive enough for making useful relative comparisons between different approaches.

For comparison purposes, we constructed a full data-capture scheduler with two 64-bit 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. All schedulers have 32 entries and are capable of issuing four instructions each cycle: two simple integer instructions, one memory instruction, and one complex integer instruction.

Table 1 shows the cycle time, as explained in Section 2.2, and the area of the different sched-

	Cycle(ns)	Energy (nJ)	Area (mm^2)
Full Data-Capt. Sched.	2.04	1.40	1.98
Narrow Data-Capt. Sched.	1.71	1.47	1.49
Narrow Data-Capt. Sched.w/ Embed. ALUs	1.28	1.48	1.53
Non Data-Capt. Sched.	1.28	1.55	1.43

Table 1: Cycle Time, Energy, and Area Comparison

uler designs. For fair comparison, the energy and the area include all logic for scheduling, operand read, and integer computation, which encompasses both payload RAM and integer ALU for the non data-capture scheduler.

As seen in Table 1, the narrow data-capture scheduler with embedded ALUs achieves the same cycle time as the non data-capture scheduler; which is 1.28 ns, and reduces cycle time by 0.43 ns as compared to naive narrow data-capture scheduler. As expected, a full data-capture scheduler requires a slow 2.04 ns cycle time.

To be conservative, a wire delay estimation using distributed RC model [26], *1/2RwireCwire*, was also done. We estimate that for non data-capture scheduler, tag broadcast has to span the width and the length of the module to reach every entry in the scheduler. Similarly, data broadcast for narrow data-capture scheduler has to also go through the width and the length of the module. Thus the wire length is width + length of each module. Assuming an ideal square layout, the area in Table 4 is used to determine the wire length. Resistance and capacitance per unit length is taken from LSI Logic's wireload model. The wire delay estimation is 0.00596 ns and 0.01484 ns for non data-capture and narrow data-capture scheduler respectively. Those results are insiginificant that narrow data-capture scheduler's cycle time is still comparable with non data-capture scheduler's.

Both narrow data-capture schedulers consumes similar amounts of power even though the one with embedded ALUs has an ALU per scheduler entry, because only the embedded ALUs that are executing are switching every cycle. Compared to the non data-capture scheduler, we save 0.07 nJ per cycle. We add 0.1 mm^2 area compared to the non data-capture scheduler; which means a slight increase in leakage power. However, Table 3 shows that this increase is insignificant compared to the overall processor leakage.

Leakage energy is evaluated by extending HotLeakage [22] to model our pipelined data cache, schedulers, payload RAM, store queue, and load queue. All leakage energy calculation is done at 70C. Leakage energy comparison is presented in Table 3. The 'Remainder' includes schedulers, payload RAM, store queue, and load queue.

3.2. Pipelined Data Cache and Store Queue

We model our pipelined data cache by modifying CACTI 3.0. [18] To avoid adding ports, the cache is separated into a narrow bank and wide bank as explained in Section 2.3. In addition to accessing the wide bank in a direct-mapped fashion, the cache bitslicing technique is employed to save additional decode power.

Table 2 shows data comparison between our pipelined data cache and 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 of which is the latency to access the narrow bank for partial tag match. This latency added to the scheduler cycle time in Table 1 is the time needed before scheduling load dependent instructions. Since narrow cache latency is less than scheduler cycle time, only one additional cycle is needed after the schedule stage to broadcast narrow data and wake up a load's dependents.

	Pipelined Data Cache	Conventional Data Cache
Latency - Narrow Bank	0.80ns	N/A
Latency - Wide Bank	1.28ns (0.6ns)	1.24ns
Energy Consumption	(0.37 + 0.22)nJ	(0.62 + 0.27)nJ
Area	(1.50 + 0.45)mm^2	(1.21 + 0.63)mm^2

Table 2: Pipelined vs Conventional Data Cache

	Narrow Data-Capture Scheduler (nJ)	Non Data-Capture Scheduler (nJ)
Remainder	0.0184	0.0182
D-Cache	0.2995	0.3161
I-Cache	0.4911	0.4911

Table 3: Leakage Energy per Cycle

The access latency of the wide bank for our cache is 1.28ns, slightly larger than the 1.24ns access latency of the conventional data cache. 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 mux drivers. However, since the index bits are available early from the narrow core, the only operations needed after the full bits are available are tag comparison and output mux drive, which requires only 0.6ns as shown in parentheses in Table 2. This exposed latency is used as the cache hit latency in our optimized proposed model, reducing the number of cycles needed to access the cache from two cycles to one. Section 4.3 explores the performance benefits from this reduction in cache access latency.

Energy consumption is also shown in Table 2 for both cache and store queue. Our cache consumes 0.37 nJ per access, significantly lower than the conventional data cache of 0.62 nJ per access. We achieve this energy improvement by disabling unneeded row decoders and by performing a direct-mapped lookup to the selected way and comparing tag for the upper address bits instead of doing set-associative access. Additional energy, 0.05 nJ, is also saved by our direct-mapped wide store queue.

The increase in the data cache area is mainly from additional interconnect as we have two instead of one cache bank and additional decoders. Though the additional decoders increase leakage energy, reductions in the number of sense amps, comparators, and mux drivers as we transform the wide bank into direct-mapped structure result in a net leakage reduction, as seen in Table 3. Leakage energy dissipated by instruction cache is included in Table 3 to show that the leakage energy dissipated by non-cache structures is not as significant.

3.3. Execution Pipeline Power Modeling

To model energy consumption and latencies for the execution pipeline, key units are designed using Verilog. The payload RAM, integer ALU, store queue, load queue, and multiplication/division unit were modeled. We assume a machine model as described in Table 5, a 32-entry narrow data-capture scheduler with embedded ALUs, matching 32-entry payload RAM, 24-entry store queue, and 32-entry load queue. We assume 64-bit operands for the non data-capture datapath and 16 bits for the narrow data-capture datapath (the multiplication/division unit is the same for all models). We model two store and load queues for our narrow data-capture datapath: narrow 16-bit

	Narrow Data-Capt. Sched		Non Data-Capt.Sched	
	Energy (nJ)	Area (mm^2)	Energy (nJ)	Area (mm^2)
Scheduler	0.68	0.62	0.49	0.25
Payload RAM	0.75	0.83	1.00	1.12
Store Queue	0.08/0.14	0.18/0.27	0.27	0.63
Load Queue	0.10/0.15	0.42/0.30	0.36	0.84
Integer ALU	0.02	0.04	0.04	0.06
Mult/Div Unit	0.43	0.35	0.43	0.35

Table 4: Energy per Access and Area

fully-associative store and load queues and wide 48-bit direct-mapped store and load queues. The premise 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 preventings loads that are known to be misses from accessing the wide store queue. The same approach is applied to the load queue. The benefit gained is similar to that of filtering schemes described in [19] and other related work on store queue scaling.

Table 4 shows the energy per access and the area for each unit. Both store and load queues for the narrow data-capture scheduler scheme have two energy and area numbers, corresponding to the narrow queues and wide queues. The energy estimate in Table 4 is incorporated in our detailed power model in Section 4.

The latencies used in Table 5 are determined using the cycle times and access times from different key units modeled. We use scheduler's cycle time as our clock period. Since our narrow data-capture scheduler achieves the same cycle time as the non data-capture baseline, the clock period used on both narrow data-capture and non data-capture machines are the same. Both payload RAM access and integer ALU execution can be done in one cycle. Hence operand read and ALU execution only need one pipeline stage each. The L1 data cache needs two cycle for access since the access latency for the conventional cache is 1.24 ns before an approximate addition of 0.2 ns for latch delay and another 0.2 ns of latch setup time (these delays correspond to the latches in the LSI Logic standard cell library we are using). To provide a fair comparison, we also use a 2-cycle cache access latency in our base scheme. Our pipeline diagram is shown in Figure 6. Our optimized machine model, presented in Section 4.3, exploits the 0.6 ns exposed wide-cache latency from Table 2 to reduce cache access to one cycle.

4 . Experimental Results

4.1. Simulated Machine Model and Benchmarks

Our execution-driven simulator was originally derived from the *Simplescalar / Alpha* 3.0 tool set [20], 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. 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. Our simulator also models aggressive load-store reordering with a memory dependence predictor similar to the Compaq Alpha 21264 machine [5]. We model a 12-stage out-of-order pipeline with 4-instruction machine width. The pipeline structure is illustrated in Figure 6. The detailed configuration of the machine model is shown in Table 5.

Out-of-order Execution	4-wide fetch/issue/commit, 64 ROB, 32 LQ, 24 SQ, 32-entry scheduler, 12-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 with highest 48 bits clock gating feature (1-cycle), 1 integer mult/div (3/20-cycle), 1 general memory ports (1+2)
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: 150-cycle latency
Store-to-Load Forwarding	Same as L1 D-Cache latency (2 cycles)

Table 5: Machine Configurations

Benchmark	Energy/Inst(nJ)	IPC	Benchmark	Energy/Inst(nJ)	IPC
bzip2	1.91	1.02	ammp	1.79	0.48
crafty	1.90	1.38	applu	1.63	0.81
eon	1.95	0.94	apsi	1.77	0.90
gap	1.89	0.68	art	1.87	0.30
gcc	1.82	1.18	equake	1.93	0.40
gzip	1.72	0.93	facerec	1.78	1.38
mcf	2.37	0.12	fma3d	1.88	0.90
parser	1.92	0.87	galgel	1.74	0.91
perlbmk	2.11	0.94	lucas	1.32	0.75
twolf	1.95	0.65	mesa	1.98	0.93
vortex	1.66	1.48	mgrid	1.82	1.40
vpr	1.95	0.75	sixtrack	1.82	1.05
			swim	1.58	2.00
			wupwise	1.67	1.44

Table 6: Benchmark Programs Simulated

Wattch [21] power models are modified and integrated in our simulator to estimate dynamic power consumption and leakage power dissipation. Energy numbers from our synthesis results are used for modeled key units while the Wattch power model, scaled to 100nm technology, is used for the other components.

The SPEC INT2000 and SPEC FP2000 benchmark suites are used for results presented in this paper. All benchmarks were compiled with the DEC C and C++ compilers under the OSF/1 V4.0 operating system using -O4 optimization. Reference input sets and SMARTS [23] statistical sampling methodology were used for both SPEC INT2000 and SPEC FP2000 results.

Throughout the evaluation, we use a non-speculative machine model as a baseline. The energy number and the IPC number for our base machine model is presented in Table 6. We compare four different machine configuration against the base machine model. The first configuration, referred as narrow, is our proposed narrow data-capture scheduler scheme with a squashing replay scheme. The second one, referred as 21264, is a non data-capture scheduler machine with a 4-bit saturating counter as a hit/miss prediction and a squashing replay scheme, similar to the Compaq Alpha

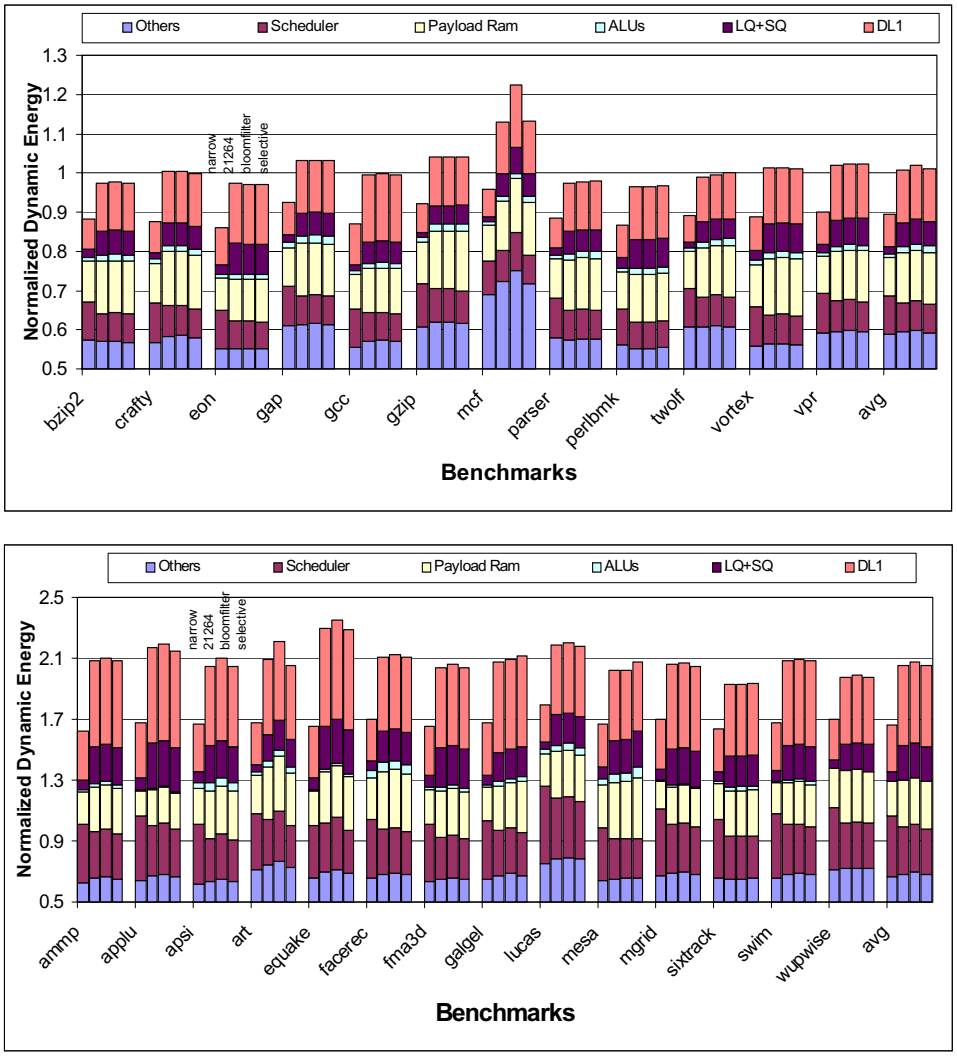

Figure 8: SPEC INT (upper) and SPEC FP (lower) Normalized Total Dynamic Energy Consumption. OoO core units include scheduler, payload RAM, load queue, store queue, L1 data cache, and ALU. Other units include other modules in microprocessor core, L1 instruction cache, result bus, and clock distribution network.

21264 machine. For the third model, referred as Bloom filter, we change the hit/miss predictor into a 64x Bloom filter predictor [8]. This third model can also represent prior work on partial tag matching [1] as the 64x Bloom predictor will give the upper bound benefit of a naive implementation of partial tag match. The fourth, referred to as selective, is the same machine as the second model, with an ideal selective replay scheme instead of squashing replay. The last three models also implement clock gating of unused 48S bits portion of integer ALUs as proposed in [14] for fair baseline comparison.

4.2. Energy Evaluation

We use a combination of a Wattch-like approach and synthesis results to estimate the dynamic energy consumed by the processor core during program execution. Energy dissipation for both SPEC INT and SPEC FP are compared in Figure 8 with four bars corresponding to the four models

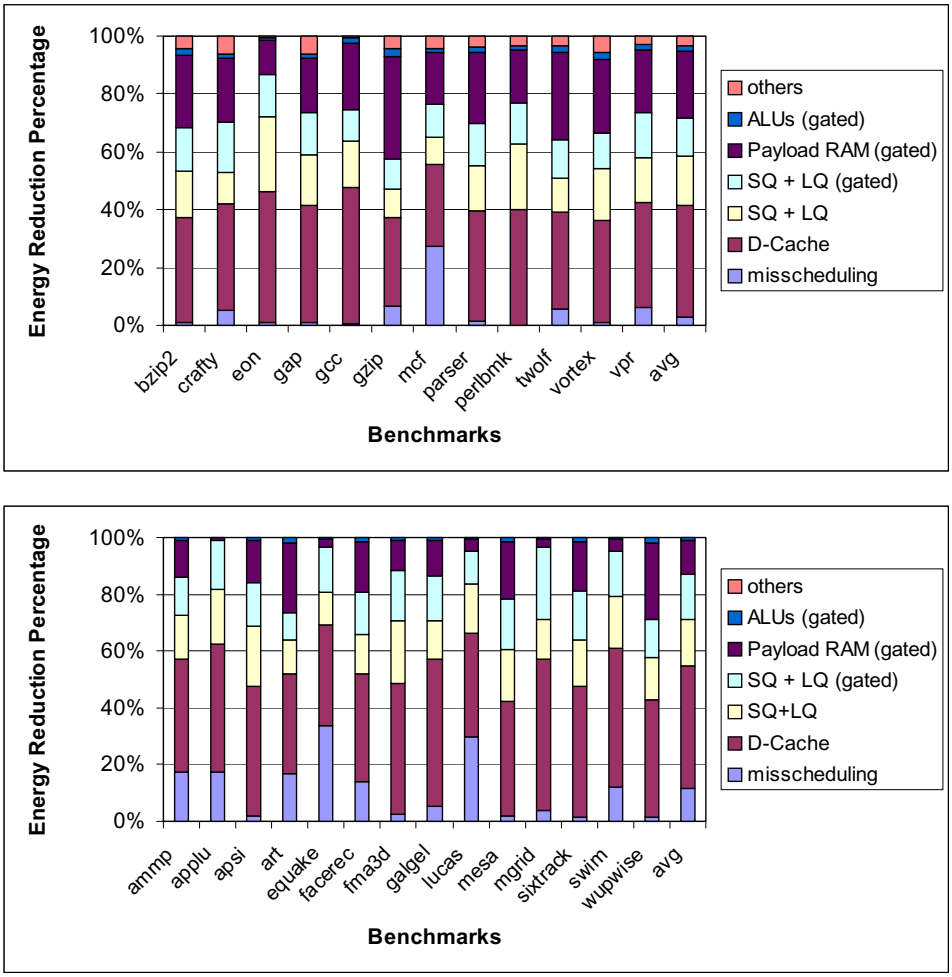

Figure 9: SPEC INT (upper) and SPEC FP (lower) Percentage of Energy Reduction in OoO core. Energy reduction of our scheme compared to 21264 machine model

described in Section 4.1. Our scheme is shown as the left-most bar.

As seen from the graph, the narrow model saves a significant amount of energy in the out-of-order core window. For SPEC INT, we save 26%-27% of out-of-order core energy compared to baseline alpha, Bloom filter, and the ideal selective recovery scheme, which translates to 11%-12% of total processor energy saving. For SPEC FP, the energy saving for the out-of-order core is 20%-21% compared to the other three bars.

The energy saving mostly comes from payload RAM, load queue, store queue, and cache. Nonetheless, since our scheduler stores partial operands and has embedded ALUs, it consumes approximately 29% more energy than non data-capture one.

Shown in Figure 8 are the energy dissipation by ALUs after clock gating [14] is applied to each model, which is about 0.8% of the entire processor power. Since ALUs energy dissipation without clock gating is approximately 1.3% of entire processor energy consumption, 66.2% of ALUs' energy saving by [14] translates into roughly 0.5% over entire processor core.

To understand the energy reduction in out-of-order core more, we divide energy reduction into seven categories as in Figure 9. Misscheduling represents reduction in issued instructions due to

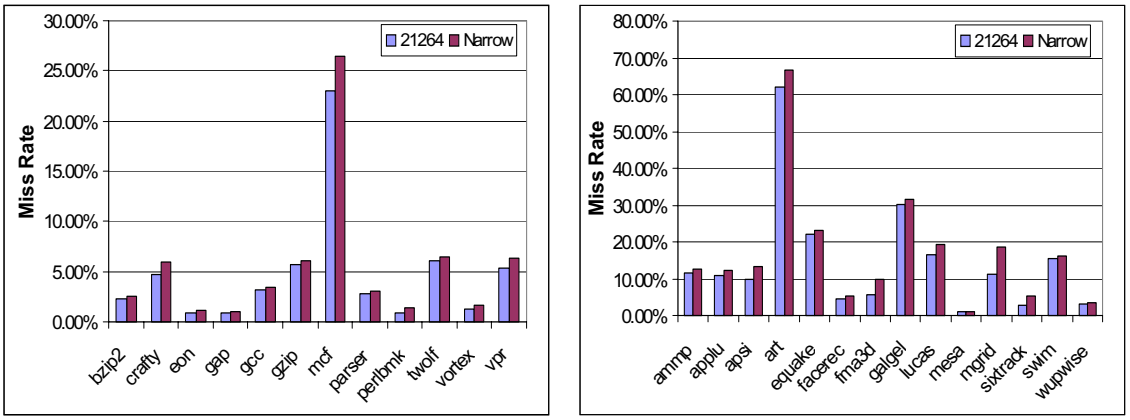

Figure 10: SPEC INT (left) and SPEC FP (right) Miss Rate (Miss/Access) Comparison.

less load misscheduling. D-Cache represents energy saving by data cache, mostly from transforming set associative wide bank into direct-mapped one. Similarly, SQ+LQ represents energy saving by changing load and store queues from fully associative into direct-mapped. The next category represents cases where there are no aliasing in narrow queues so that the wide queues can be gated or disabled. The Payload RAM category includes disabling of payload RAM for narrow operation; the same applies for ALUs category. Finally, others shows the benefit of reducing the wide data path from 64 bits to 48 bits.

For SPEC INT, it is shown that approximately 56% of energy reduction comes from transforming cache and queues into direct-mapped ones. Disabling wide portion of load and store queues for non-aliasing cases adds 13% of saving on average. 22% of the reduction comes from gating the payload RAM accesses for narrow operation cases while energy saving from disabling the ALUs is about 2%. Reduction in misscheduling instructions only adds about 4% of energy saving. The reason for this is that the baseline that we compared to already has a 21264-like counter hit/miss predictor in addition to the already low data cache miss rate, except for mcf.

We can also see in Figure 9 that SPEC FP benchmarks have a similar energy reduction breakdown with SPEC INT benchmarks. The two main differences are that SPEC FP benchmarks have lower energy reduction from gating payload RAM and ALUs and higher energy reduction from misscheduling category. It is expected since SPEC FP has more instructions that are scheduled non-speculatively , thus less payload RAM gating. The fact that SPEC FP benchmarks have higher miss rates as shown in Figure 10 gives a better opportunity to reduce energy consumption due to load misscheduling.

Miss rate comparison for both SPEC INT and SPEC FP can be seen in Figure 10. Excluding mcf, the SPEC INT average miss rate is 3.11% and 3.56% for the baseline 21264 model and narrow scheme respectively; a 14% increase in miss rate for the narrow scheme compared to the baseline 21264 model. For mcf, the miss rate is 24% and 26.5% for each scheme; it explains the significant energy reduction from misscheduling category. The slight increase in miss rate in our narrow scheme comes from the fact that we do not allow narrow tag aliasing in our data cache, leading to additional conflict misses. However, the increase in miss rate does not affect the performance so much as can be seen in Section 4.3 due to the fact that the miss rate itself is already low and that our narrow scheme successfully eliminated a large number of replays.

For SPEC FP, the average miss rate is 11.14% and 13.21% for the baseline 21264 model and the narrow scheme respectively. For art, the miss rate 61.94% and 66.82% for each scheme respec-

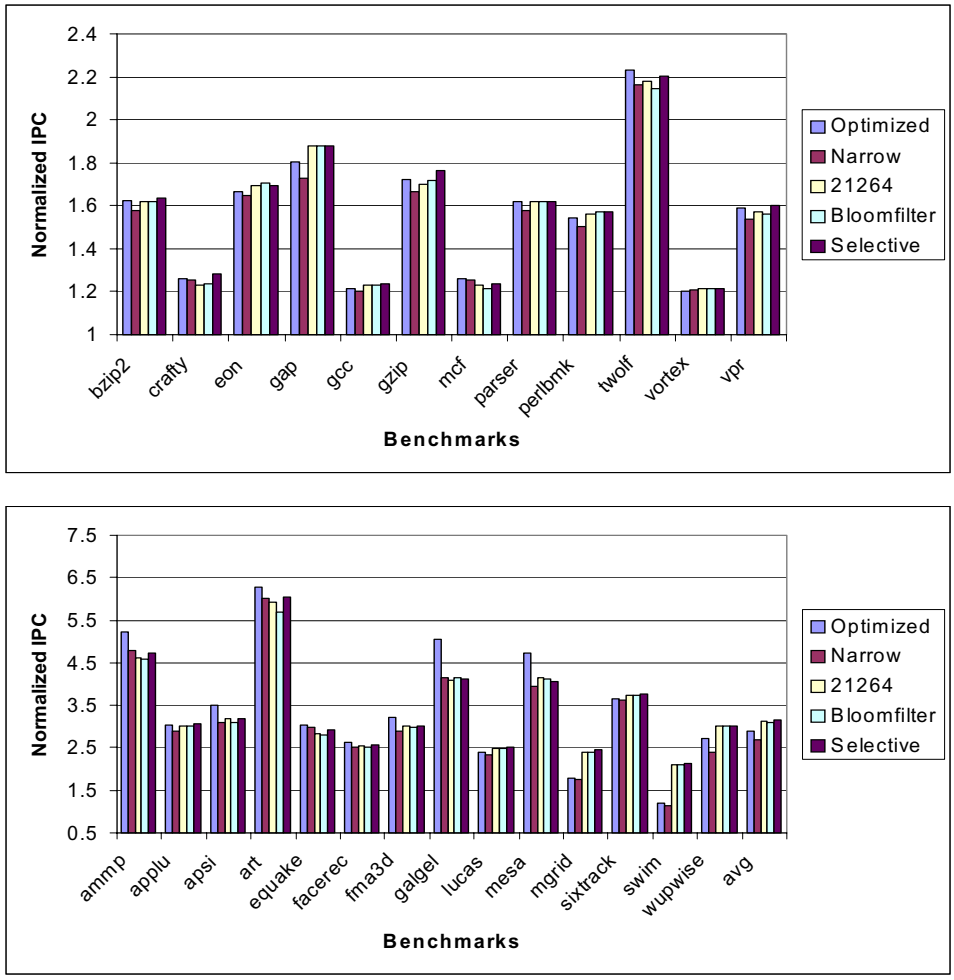

Figure 11: SPEC INT (upper) and SPEC FP (lower) Optimized Narrow Data-Capture Scheduler with Refetch Replay Scheme.

tively. Overall SPEC FP benchmarks have higher miss rate than SPEC INT benchmarks, which explains the reason why SPEC FP benchmarks get a higher energy reduction from misscheduling reduction than SPEC INT benchmarks.

4.3. Performance Evaluation and Optimization

Unlike other power saving schemes that often cause significant reduction in performance, our scheme has a minimal effect. Figure 11 shows the comparison of normalized IPC number among the four machine models described. Our scheme is the second bar from the left. Compared to the 21264 and Bloom filter scheme, our narrow scheme only loses by 1% for SPEC INT. This performance hit is caused by the fact that we have to do non-speculative scheduling for complex ALU instructions and unaligned loads. The performance of our scheme is only 2% less compared to the ideal selective replay scheme for SPEC INT benchmarks.

For fair comparison, performance evaluation for the second bar from the left assumed the same pipeline length and scheduler entry release policy for all options. However, we also evaluate the performance of a narrow data-capture scheduler with refetch replay scheme where these policies are improved to match attributes of our scheduler architecture. As our scheme eliminates most

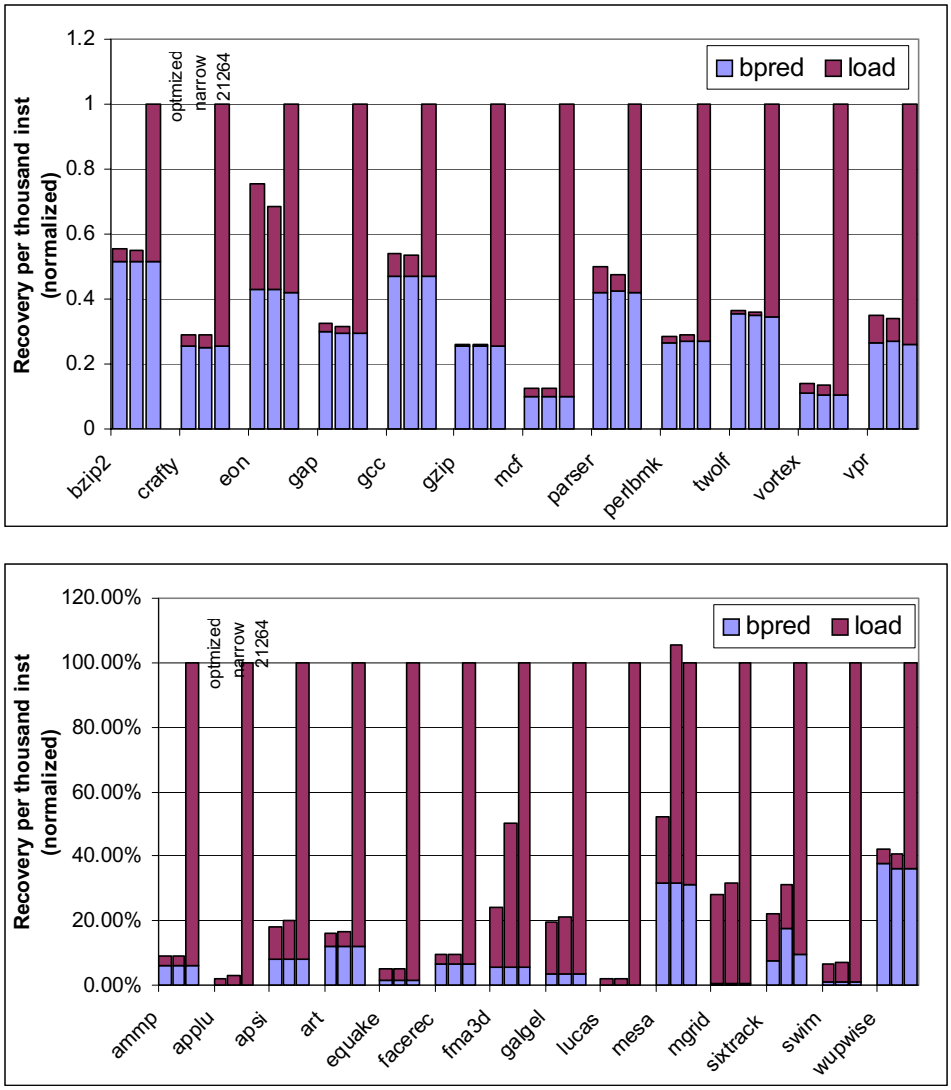

Figure 12: SPEC INT (upper) and SPEC FP (lower) Number of pipeline flush recovery due to branch misprediction and load misscheduling. Those numbers are normalized to 21264 model.

of the load mis-scheduling, a simple refetch replay scheme can be used without sacrificing substantial performance. Because a refetch replay scheme can use the existing recovery structure for branch mispredictions, the complexity of the machine can be significantly reduced.

We exploit the fact that, using a refetch replay scheme, it is unnecessary to keep the instructions in the scheduler once they are issued, since a misscheduling event will cause them to be refetched by the front end. This early issue slot reclaim means that 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, referred as optimized scheme, we assume one cycle exposed cache access latency rather than two cycles latency. As shown in Table 2, this is a reasonable assumption. Our optimized

model is shown as the left most bar in Figure 11. As shown, for SPEC INT, it can recover the performance loss due to the non-speculative scheduling of certain instructions. In fact, our optimized scheme performs comparably with the ideal selective scheme.

SPEC FP performance is not as good as SPEC INT suites. On average, the naive narrow scheme performance is 14% less compared to alpha 21264 model while the optimized scheme performance is 7% less. In fact our scheme fails terribly on mgrid and swim due to the fact that both bencmarks have a fair amount of non-speculatively scheduled instructions.Additionally, the new cache replacement policy increases the miss rate for mgrid by 64%. In this case, a small victim cache might help reduce the increased in miss rate.

Figure 12 shows the number of recovery caused by branch misprediction and load misscheduling per thousand of instructions. From left to right are the optimized refetch scheme, the naive narrow scheme, and the 21264 alpha model. Those numbers are normalized to 21264 alpha model. From Figure 12 it can be seen that narrow scheduling scheme is able to reduce the number of replay/recovery caused by load misscheduling for SPEC INT significantly, except for eon. It is also important to notice that for the optimized refetch scheme, load misscheduling only adds 5%-8% pipeline flush to the existing branch misprediction flush.

Similarly, for SPEC FP benchmarks, our narrow scheme also reduces load misscheduling recovery significantly. However, due to the larger portion of load misscheduling recovery to branch misprediction recovery for SPEC FP benchmarks, the optimized refetch scheme adds a fair amount of pipeline flush to the existing branch misprediction flushes.

5 . Background and Related Work

Our background and previous work are classified into two sub-sections. We discuss previous work on speculative scheduling in Section 5.1. Section 5.2 describes previous work on exploiting narrow-width operands to achieve higher performance and reduce power consumption

5.1. Speculative Scheduling

Different recovery mechanisms for speculative scheduling have been proposed, each with different complexity and performance impact. The simplest one is refetch recovery, where all misscheduled instructions are flushed and refetched. The Compaq Alpha 21264 [5] uses squashing recovery, where all instructions scheduled after the misscheduled load are canceled and replayed. A more sophisticated recovery mechanism called serial selective replay is implemented in Intel Pentium 4 [5]; here, only instructions dependent on the misscheduled load are replayed. However, this mechanism is unable to stop the propagation wavefront that keeps waking up more dependent instructions immediately. An ideal recovery mechanism would be able to stop the propagation wavefront soon after a misscheduling is detected and reschedule only the dependent instructions. Nonetheless, this ideal scheme is too complex to implement in current designs [6]. In Section 4.3, we show that a simple refetch recovery scheme is sufficient since we are able to eliminate virtually all sources of misscheduling.

Since rescheduling of instructions wastes power and execution bandwidth, several techniques have been proposed and even implemented to reduce the number of misspeculated loads. The Compaq Alpha 21264 [5] uses the most significant bits of a 4-bit saturating counter as a load's hit/miss prediction. The counter is incremented on a cache hit and decremented on a cache miss, and predicts the likelihood of a cluster of cache misses. 2-level local predictors, 2-level global predictors, and hybrid predictors are proposed by Yoaz et al. [7] for cache hit/miss prediction. The most recent work on predicting cache hit/miss behavior is by Peir et al. [8]. Peir uses a Bloom filter that is indexed using a partial address to determine whether the load will hit the cache. The filter is updated everytime a cache miss is encountered.

The other uncertainty to be resolved for speculative load scheduling is whether a load should be deferred until all earlier stores have generated their addresses. The Compaq Alpha 21264 aggressively executes loads past prior unresolved stores and sets a flag in a memory dependence predictor table for the loads that are incorrectly issued. Hesson et al. of IBM filed a patent for the store barrier cache [9], which keeps track of stores that tend to cause memory-order violations and inhibits all subsequent loads from executing pass that violating stores. Moshovos et al. [10] uses a pair of fully-associative structures that hold store-load pairs that have caused memory-order violation. This structure will then be used to direct the order of execution for subsequent instances of the instructions. Similar work was done by [11]. In [12], Moshovos and Sohi use memory dependence prediction to guide forwarding of data values from stores to dependent loads. Chrysos and Emer [13] use two tables, one for associating loads and stores into sets and the other to track the use of these sets, to help find misordered store-load pairs.

5.2. Narrow Width Operands

Exploitation of narrow-width operands has been studied repeatedly in the last decade. One of the first to exploit narrow-width operands to save power in the execution unit is Brooks et al. [14]. He proposes to aggressively clock gate the unused portion of integer functional units when a narrow-width operand is encountered. Brooks also proposes to merge narrow integer operations and to allow them to share a single functional unit. A more aggressive approach by Canal et al. [15] compresses data, addresses, and instructions by maintaining only significant bytes with extension bits appended to indicate the significant byte positions. The extension bits are later used to only enable necessary pipeline operations.

Liu et al. [1] proposes cache designs with partial tag match to enable early hit/miss determination. Partial tag bits are first checked and a hit/miss prediction signal is generated. A hit/miss signal confirmation is sent after the full tag bits are compared. However, partial tag match alone is not able to reduce misscheduling penalty significantly since it only generates a partial miss signal a cycle or two earlier than the full tag miss signal.

Mestan et al. [2] more aggressively exploits partial operand knowledge of input operands to overlap the execution of dependent instructions and resolve unknown dependences. The processor's datapath is sliced into 8-bit or 16-bit slices to fully exploit the operand knowledge for load-store disambiguation, partial cache-tag matching, and early branch misprediction detection.

A pipelined cache design that exploits partial operand knowledge to save decoding power is proposed in [16]. Early partial address bits are used to determine and disable row decoders not used on that access.

6 . Conclusions and Future Work

This paper proposes a novel dynamic scheduling mechanism that uses partial operand knowledge to reduce power consumption due to unnecessary speculative activities in a microprocessor. Specifically, the out-of-order core is sliced into a narrow core and wide core. The narrow core operates on the least significant sixteen bits of each operand to perform early load-store disambiguation and partial tag matching. The results are used to avoid load mis-scheduling and to enable direct-mapped access of data cache, store queue, and load queue in the wide core.

Power consumption is reduced dramatically by eliminating unnecessary speculative activity, including speculative issue of cache-miss-dependent instructions, and associative look-ups of data cache, store queue, and load queue in the wide core. Additional power is saved when both operands are narrow, since no access is needed for payload RAM and ALU. Approximately 26%-27% of the out-of-order core's energy can be saved, which translates to roughly 11% of total processor core energy for SPEC INT benchmarks. For SPEC FP benchmarks, the out-of-order core's energy

reduction is roughly 20%. In addition to reducing dynamic power, our scheme also reduces leakage power in the L1 data cache. All the power savings above are accomplished with virtually no performance penalty. On average, we only lose 1% and 14% performance compared to the non data-capture machine with the Alpha 21264 counter hit/miss predictor for SPEC INT and SPEC FP benchmarks respectively.

Since our technique can eliminate most of misspeculated loads, less complex recovery mechanism can be employed without losing significant performance. With narrow data-capture scheduler, a simple refetch replay scheme performs comparably with a complex squashing replay scheme combined with a sophisticated Bloom filter hit/miss predictor. This optimized scheme effectively recovers most of the performance lost due to non-speculative scheduling of complex instructions and unaligned loads.

We believe that further benefits can be reaped by our narrow data-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 can be completed in the schedule stage. The embedded ALUs also provide hardware redundancy, which could be exploited further for transient or permanent fault detection.

Acknowledgements

We thank the anonymous reviewers for their comments and suggestions. This research was supported in part by the National Science Foundation under grants CCR-0133437 and CCF-0429854, as well as grants and equipment donations from IBM and Intel.

References

[1] L.Liu, Cache Designs with Partial Address Matching, In *ISCA-27,*1994.

[2] B.Mestan, and M.H.Lipasti, Exploiting Partial Operand Knowledge, In *ICPP*, 2003.

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

[4] J.P.Shen and M.H.Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processor*, beta edition, pp.178-181. McGraw-Hill, 2002.

[5] R.E.Kessler et al., The Alpha 21264 Microprocessor Architecture, In *ICCD*, 1998.

[6] I.Kim and M.H.Lipasti, Understanding Scheduling Replay Schemes, In *HPCA-10*, 2004.

[7] A.Yoaz et al., Speculation Techniques for Improving Load Related Instruction Scheduling, In *ISCA-26*, 1999.

[8] J.K.Peir et al., Bloom Filtering Cache Miss for Accurate Data Speculation and Prefetching, In *ICS*, 2002.

[9] J.Hesson et al., Apparatus to Dynamically Control the Out-Of-Order Execution of Load-Store Instructions, *US. Patent 5,615,350*, Filed Dec. 1995, Issued Mar. 1997.

[10] A.Moshovos et al., Dynamic Speculation and Synchronization of Data Dependences, In *ISCA-24*, 1997.

[11] G.Tyson and T.Austin, Improving the Accuracy and Performance of Memory Communication Through Renaming, In *Micro-30*, 1997.

[12] A.Moshovos and G.Sohi, Streamlining Inter-opration Memory Communication via Data Dependence Prediction. In *Micro-30*, 1997.

[13] G.Chrysos and J.Emer, Memory Dependence Prediction using Store Sets, In *ISCA-25*, 1998.

[14] D.Brooks and M.Martonosi, Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, In *HPCA-5*, 1999.

[15] R.Canal et al., Very Low Power Pipelines using Significance Compression. In *Micro-33*, 2000.
[16] E.Gunadi and M.H.Lipasti, Cache Pipelining with Partial Operand Knowledge, In *WCED*, June 2004.
[17] K.Flautner et al., Drowsy Caches: Simple Techniques for Reducing Leakage Power, In *ISCA*, 2002.
[18] P.Shivakumar and N.P.Jouppi, CACTI 3.0: An Integrated Cache Timing, Power, and Area Model.
[19] S.Sethumadhavan et al., Scalable Hardware Disambiguation for High ILP Processors, In *Micro-36*, 2003.
[20] D.C.Burger and T.M.Austin, The Simplescalar tool set, version 2.0, *Technical Report CS-TR-97-1342*, UW-Madison, 1997.
[21] D.Brooks et al., Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, In *ISCA-27*, 2000.
[22] Y.Zhang et al, HotLeakage: An Architectural, Temperature-Aware Model of Subthreshold and Gate Leakage, *Technical Report CS-2003-05*, Department of Computer Sciences, University of Virginia, March 2003.
[23] R.E.Wunderlich et al., SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling, In *ISCA-30*, 2003
[24] N. Richardson, Lun Bin Huang, R. Hossain, J. Lewis, T. Zounes, and N. Soni, The iCore 520-MHz synthesisable CPU core, *IEEE MICRO*, vol. 23, no. 3, 2003 pp. 46 - 57.
[25] R. Ho, K.W. Mai, M.A. Horowitz, The Future of Wires, *Proceedings of the IEEE*, vol. 89, no. 4, April 2001, pp. 490 - 504.
[26] H.E. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 3rd edition, May 2004.