Implementing Optimizations at Decode Time

Ilhyun Kim and Mikko H. Lipasti
Dept. of Electrical and Computer Engineering
University of Wisconsin–Madison
ikim@cae.wisc.edu, mikko@ece.wisc.edu

Abstract

The number of pipeline stages separating dynamic instruction scheduling from instruction execution has increased considerably in recent out-of-order microprocessor implementations, forcing the scheduler to allocate functional units and other execution resources several cycles before they are actually used. Unfortunately, several proposed microarchitectural optimizations become less desirable or even impossible in such an environment, since they require instantaneous or near-instantaneous changes in execution behavior and resource usage in response to dynamic events that occur during instruction execution. Since they are detected several cycles after scheduling decisions have already been made, such dynamic responses are infeasible. To overcome this limitation, we propose to implement optimizations by performing what we call speculative decode*. Speculative decode alters the mapping between user-visible instructions and the implemented core instructions based on observed runtime characteristics and generates speculative instruction sequences. In these sequences, optimizations are prescheduled in a manner compatible with realistic pipelines with multicycle scheduling latency. We present case studies on memory reference combining and silent store squashing, and demonstrate that speculative decode performs comparably or even better than impractical in-core implementations that require zero-cycle scheduling latency.*

1. Introduction

Program compilation and optimization consists of a sequence of semantic bindings that bridge the gap between high-level programming languages and the hardware primitives used to implement their semantics. These bindings can occur early, as in a static compiler (Figure 1a) that creates an optimized binary by exploiting high-level program knowledge and global analysis to remove redundant code, perform common subexpression elimination, assign variables to registers, and so on. Semantic binding can also occur later in the program's life-cycle, during program load time with a just-in-time (JIT) compiler (Figure 1b), or even as a set of peephole optimizations performed at near runtime on the program code residing in a processor's trace cache (Figure 1c) [2]. Finally, optimizing transformations that affect or determine which hardware primitives are employed to realize program semantics can even be applied within the processor's execution pipeline (Figure 1e). The degree and accuracy of dynamic information that is available to guide such optimization increases as the process of semantic binding is delayed until right before the hardware primitives are executed. It is these types of late run-time optimizations that interest us; ones that both require dynamic knowledge of program execution characteristics in order for them to be fruitful, and are difficult to implement earlier in the program's lifetime.

In this paper, we argue that technology and implementation trends make it impractical to delay the semantic bindings that implement such run-time optimizations any further than a processor's decode stage. In classic out-of-order processors based on Tomasulo's algorithm [21], dynamic events observed during instruction execution could be used as inputs into the scheduling process to affect scheduling decisions for the very next cycle. As shown in Figure 2, modern out-of-order processors like the Alpha 21264 [14] and Intel Pentium4 [16] implement deep execution pipelines that separate dynamic instruction scheduling from instruction execution by several stages. Unfortunately, the *scheduling latency* induced by these additional pipeline stages in modern designs prevents such instantaneous feedback, since the execution schedules have to be created several cycles in advance. As a result, such pipelines cannot feasibly implement run-time optimizations that derive benefit from the processor's ability to immediately react to observed events and reassign execution resources for the very next execution cycle.

Given these implementation trends, run-time optimization inside the dynamic execution window is no longer practical; in our view, the most feasible time for implementing run-time optimizations is in the decode stage (Figure 1d). In this paper, we propose implementing dynamic optimizations at decode time without perturbing the dynamic scheduling logic by performing what we call *speculative decode* (SD). SD alters the mapping between user-visible and implemented-core instructions based on observed runtime characteristics, and generates an instruction stream that directly expresses the run-time optimization.

SD exploits the translation layer that often exists to bridge the gap between the user-visible architected instruction set (*U-ISA*) and a realizable implementation instruction set (*I-ISA*). A number of working examples that perform translations between U-ISA and I-ISA exist, ranging from trap-based translators where unimplemented instructions are emulated in the operating system's invalid instruction exception handler (e.g. MicroVAX, PowerPC 604); to microcoded emulation routines stored in an on-

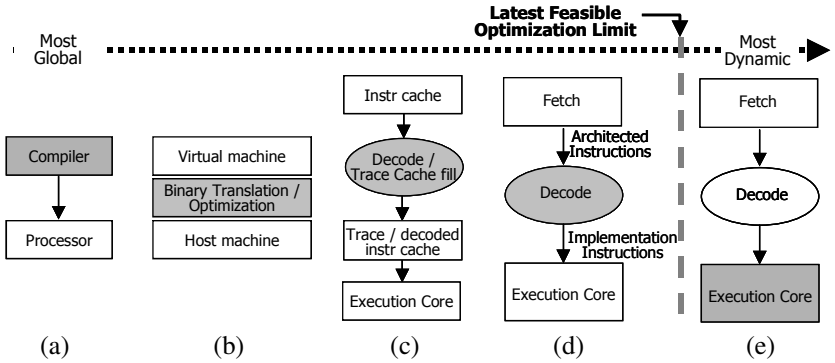


Figure 1. Optimizations in various layers. Although optimizations in the execution core may benefit from the most dynamic information, it becomes impractical with the introduction of speculative scheduling

chip lookup table (e.g. Intel Pentium Pro and it's derivatives and IBM Power4); to software-based binary translation approaches (e.g. Crusoe's Code Morpher).

This paper proposes and evaluates SD as a feasible mechanism for implementing late, run-time optimizations. Section 2 describes the difficulties in implementing in-core optimizations in a realistic processor pipeline, and introduces SD to overcome those difficulties. In Section 4, we present two case studies that address inefficiencies in the handling of memory instructions by performing speculative transformations. Section 5 provides a detailed performance evaluation of these case studies and shows that SD can reap many of the benefits and at times exceed the performance of impractical pure hardware implementations for a set of SPEC95 and SPEC2000 integer benchmarks.

2. Speculative decode

2.1. Speculative scheduling overview

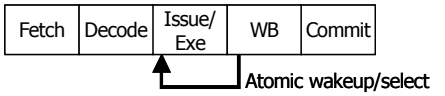
Out-of-order processors are based on Tomasulo's algorithm [21], in which instructions that finish execution wake up their dependent instructions and the scheduling logic selects issue candidates from the pool of ready instructions. As shown in Figure 2a, this atomic wakeup/select process occurs in parallel with instruction execution. In recent designs, the number of pipeline stages between instruction issue and execution has increased to accommodate the latency needed for reading the register file and performing other book-keeping duties. As instruction issue and execution stages are further separated, a naive implementation would fail to achieve maximum ILP because back-to-back execution of dependent instructions is no longer possible. To address this problem, current-generation processor implementations [14][16] use *speculative scheduling* in which the scheduler speculatively wakes up and selects dependent instructions assuming the parent instruction has a fixed execution latency, as illustrated in Figure 2b. Since load latency is not deterministic, dependent instructions are scheduled assuming the common case cache hit latency for the parent load. If the load latency is mispredicted due to e.g. cache misses, load-dependent instructions that have issued within the *load shadow* [13] will get incorrect values, so they must be *replayed* [16] with correct inputs. Of course, structural hazards (e.g. cache ports and functional units) are also resolved speculatively, based on fixed latency and common-case resource needs. Once an instruction leaves the scheduler, all operations needed for its execution are performed in a lock-step fashion that prevents dynamic changes in instruction behavior and resource allocation.

2.2. Problems with in-core optimizations and speculative scheduling

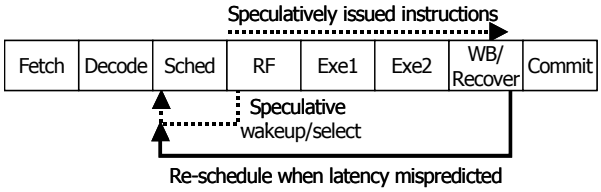
Run-time optimizations are performed in the execution core based on observed dynamic behavior. Instructions that are control, data, or resource-dependent on the optimization target benefit since they are able to execute sooner. However, if the optimizations require run-time knowledge that is not available until the optimization target is in the execution window, it will be too late to benefit these dependent instructions. For example, consider an optimization like memory reference combining [9] that avoids using a cache port, thus making it available to a subsequent instruction. If the scheduler is unaware that the memory reference is going to be combined when it is constructing the execution schedule for the pipeline, it cannot reassign the cache port to another instruction. Hence, avoiding use of the port provides no direct performance benefit. Figure 3a illustrates such a scenario, in which none of the instructions issued in the cycles following the optimization target are able to benefit from the optimization since they have already been scheduled to behave as if the optimization had not been performed.

In general, in-core implementations of microarchitectural optimizations become undesirable or even impossible in a processor that implements speculative scheduling if they depend on any of following attributes:

- *Instant re-execution*: Any technique that assumes instant re-execution of dependent instructions becomes considerably less efficient. One example that assumes instant re-execution is the next-cycle selective reissue mechanism used in value prediction and other speculative techniques.
- *Variable execution latency*: If a technique implies vari-

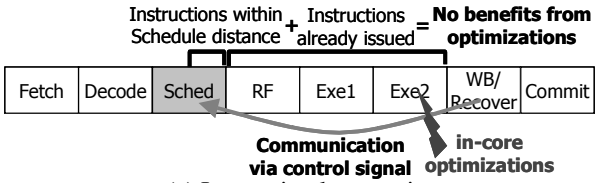


(a) Generic OoO pipeline w/ atomic wakeup / select

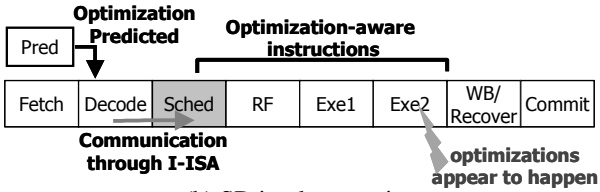


(b) 8-stage OoO pipeline w/ speculative scheduling

Figure 2. Speculative Scheduling.



(a) In-core implementations



(b) SD implementations

Figure 3. Different approaches to optimizations.

able instruction latency, dependent instructions--which must be scheduled with fixed latency--cannot benefit. If the latency assumed is longer than the actual latency, there is no benefit from the reduced latency; if the actual latency is longer, recovery is required (similar to handling a load miss). One example of variable execution latency is a sequential-associative cache [15].

- *Instant resource allocation/deallocation*: It is impractical to dynamically re-allocate processor resources once they are scheduled. For instance, store/load forwarding or memory cloaking [17] does not reduce load port contention unless it is detected and scheduled beforehand because another load cannot fill the deallocated slot.

2.3. Enabling in-core optimizations via speculative decode

We propose using speculative transformations at decode-time to overcome the challenges of implementing run-time in-core optimizations in processors with realistic scheduling latency. The key idea for enabling such optimizations is to communicate them directly to the processor core via a different sequence of instructions that express the run-time optimization explicitly, as shown in Figure 3b. Of course, the dynamic events that guide such optimizations must be predicted, since the decision to optimize must be made before the instructions leave the scheduler.

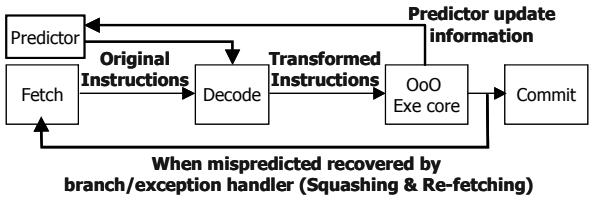


Figure 4. SD integrated into a pipeline.

Figure 4 illustrates the basic concept of SD. The predictor is trained by signals from the core that record dynamic events pertinent to the optimization at hand. If an optimization condition is predicted based on observed runtime characteristics, the decode logic transforms the architected instruction into one or more implementation instructions that express the optimization explicitly. Misprediction is detected by verification code inserted into the transformed sequence and the correct architected state is recovered by draining the pipeline, fetching the original instructions, and executing them without any speculative transformations.

Increasing the complexity of the decode logic to support such translation may seem difficult to justify. However, since translation between U-ISA and I-ISA already occurs in many processors, we believe that the current and future generation decoders will be able to incorporate SD without significantly affecting processor cycle time. For example, in an experimental S/390 processor [12] that applies a table look-up approach for instruction translation, some instructions have more than one look-up entry mapped to different sequences depending on architectural states. Moreover, existing decoders such as those for IA-32 [16] and Power4 [18] already perform one-to-multiple instruction transformations similar to the transformations proposed here. However, if a processor does not have pre-existing translation stages, SD becomes less attractive since the added complexity in the decode stage may negatively affect overall performance by either increasing cycle time or requiring additional pipeline stages. We study sensitivity to pipeline depth in Section 5.4.

In summary, the potential benefits of SD are 1) it pre-schedules optimizations in a manner compatible with realistic pipelines with multicycle scheduling latency. It does not negatively affect instruction scheduling since the optimization is transparent to the scheduler. 2) It reuses existing data path and resources in the processor core if an optimization is implemented using existing I-ISA instructions. This helps to reduce the cost of processor core redesign for new optimizations. 3) It reduces resource and queue contention better than an in-core implementation since SD affects all stages from schedule to commit.

3. Simulation Environment

3.1. Base machine model

Our execution driven simulator is based on *SimpleScalar* [6]. We model a detailed 8-stage out-of-order processor with speculative scheduling, as shown in Figure 2b. The base machine configurations are shown in Table 1. For the

translation layer in which SD is performed, we assume decode-time translation as shown in Figure 1d. We also modeled the degraded processing bandwidth due to SD by assuming that speculatively decoded instructions consume decode/dispatch/issue/commit bandwidth when an instruction is decoded into multiple operations.

3.2. Benchmark programs

We used eight benchmark programs from SPECINT95 and five from SPECINT2000 as presented in Table 2. All binaries are compiled by the gcc-pisa compiler with maximum optimizations (-O3).

4. Implementing Optimizations via SD

In this section, we discuss two possible optimizations via SD: memory reference combining [9] and silent store squashing [5]. Earlier proposals have described pure hardware implementations for these techniques. We describe how these techniques are implemented via SD, and also compare them with aggressive, even unrealistic, in-core implementations.

Table 1: Machine configuration.

Out-of-order Execution	8-stage, 4-wide fetch/issue/commit, 64-entry RUU, 32-entry load / 16-entry store schedulers, speculative scheduling, replays load-dependent instructions when load misses, fetch stops at first taken branch in a cycle
Branch Prediction	Combined bimodal (4k entries) / gshare (4k entries) with a selector (4k entries), 16 RAS, 1k-entry 4-way BTB, at least 8 cycles taken for misprediction recovery
Memory System (latency)	64KB 2-way 32B line IL1 (2), 64KB 4-way 16B line DL1 (2), 512KB 4-way 64B line unified L2 (8), main memory (50), 2 store buffers outside the OoO core
Functional Units (latency)	4 integer ALUs (1), 2 floating ALUs (2), 2 integer MULT/DIV (3/20), 2 floating MULT/DIV (4/12), 2 load ports, 1 store port, load/store ports are mutually exclusive

Table 2: Benchmark programs tested.

Benchmarks	Input sets	Instruction count
compress	compress.in	35.7M
gcc	genoutput.i	58.3M
go	go.in	85.6M
ijpeg	tinyrose.ppm	74.8M
li	queen6.lsp	41.7M
m88ksim	m88ksim.in	100M
perl	trainscrabbl.in	40.5M
vortex	vortex.in	65.1M
bzip	input.random	4.5B
gzip	input.compressed	1.3B
mcf	mdred.in	601M
parser	parsertest.in	3.1B
vpr	net.in, arch.in	1.5B

4.1. Memory reference combining

SD memory reference combining converts multiple narrow references into a single wider reference, resulting in a net reduction of accesses to the data cache. This technique is enabled by the presence of wide data paths in support of instruction set extensions that have now been added to many general-purpose instruction sets, as either a 64-bit execution mode or 128-bit media processing instructions. Despite these 64/128-bit extensions, it is expected that the vast majority of user-mode and even kernel code will continue to execute in 32-bit mode. SD memory reference combining enables existing binaries to benefit from wider memory datapaths without recompiling them. Furthermore, such an approach could logically be extended for wider reference combining up to the full cache line provided that wider data paths become available.

4.1.1. In-core implementations

There are numerous proposals to utilize a wide data path and achieve higher memory bandwidth by exploiting spatial locality in cache lines and satisfying multiple requests with a single cache access. Figure 5 shows a possible load queue modification that combines multiple loads to the same doubleword, similar to [9]. In this configuration, each load queue entry has a buffer to hold 64-bit data from memory until the entry is committed. When a load value is available from a previous entry, the value is forwarded from the buffer. 128-bit load combining can be implemented by widening the datapath from cache and increasing the buffer size. Store combining cannot be implemented this way since it could commit stores out of order. Instead, we perform store combining by allowing write merging in the store buffer [19].

Although this scheme appears to be effective in reducing cache port contention and improving load latency, there are real problems in implementing hardware load reference combining in a pipeline with multicycle scheduling latency. To obtain benefit in such a pipeline, the piggybacked (i.e. combinable) loads must be identified at schedule time so they can avoid allocating a cache port and hence their dependent instructions can be scheduled for reduced latency. However, since effective addresses are not available at schedule time, combinability cannot be determined, and all loads must allocate a cache port and schedule dependent instructions assuming the longer cache hit latency. As a result, reference combining that avoids using

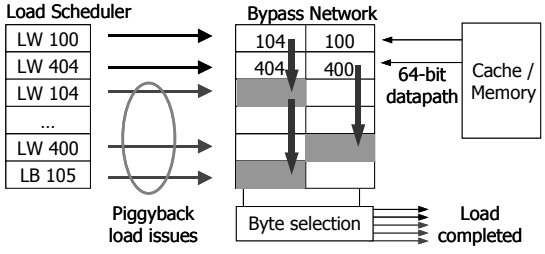


Figure 5. A possible in-core implementation for load combining.

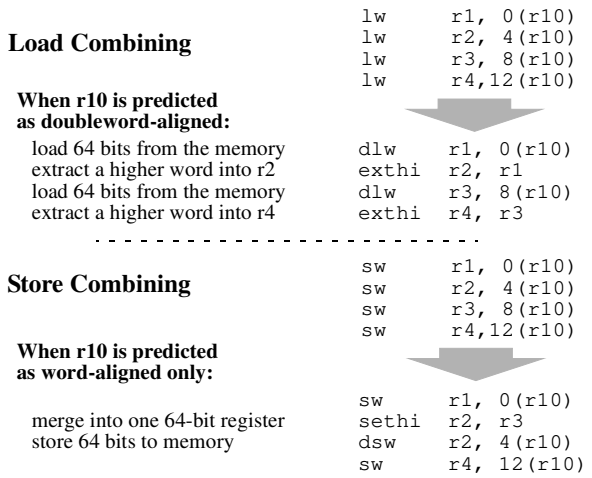


Figure 6. Examples of double-word SD memory reference combining.

the allocated cache port provides no performance benefit, and its benefit is reduced strictly to reducing power consumption by eliminating accesses to the data cache.

4.1.2. Memory Reference Combining via SD

In memory reference combining via SD, loads and stores that access consecutive memory locations are explicitly merged into one wider memory instruction. The advantage of explicit conversion is 1) it reduces load/store scheduler contention since it dispatches fewer memory instructions, and 2) it moves combining decisions from the critical path outside the out-of-order execution window. For simplicity of presentation, we only describe double-word (64-bit) combining; quad-word (128-bit) combining is implemented in an analogous manner. Performance results for both are presented in Section 5.2.

Reference Combining Mechanism. Figure 6 shows examples of double-word combining. For load combining, two loads that access consecutive memory locations are merged into one double-word load operation. An extract instruction (`exthi`) is inserted after a double-word load (`dlw`) to put the higher word into the target register. The decoded sequence can vary depending on ISAs. For example, as long as the higher 32 bits loaded in the register are ignored when a 64-bit implementation of PowerPC architecture is running in 32-bit mode, the sequence shown in Figure 6 works as illustrated. In the MIPS architecture, a double-word load instruction puts a 64-bit value into two logically adjacent registers so the decoder logic could manipulate the rename table to put values into arbitrary target registers without any ALU operations to extract the high word value. In our simulations, we assume that one load and one ALU operation are needed. Store combining can be accomplished in an analogous manner. One ALU operation (`sethi` in the example) to merge two 32-bit values into a 64-bit register and one double-word store operation are needed for store combining. Wider reference combining may either utilize datapaths and storage that exist for media processing instructions (e.g. PowerPC

Table 3: History bit patterns and their corresponding predictions in double-word combining.

History bit pattern (rightmost is newest 1:aligned 0:unaligned)	Prediction	Description
1 1 1 1	Combine	Effective address is very likely to be doubleword-aligned
1 0 1 0	Combine	Base register value is changing by word-aligned amounts
Other patterns	Do not Combine	Cannot predict the alignment or unlikely to be doubleword-aligned

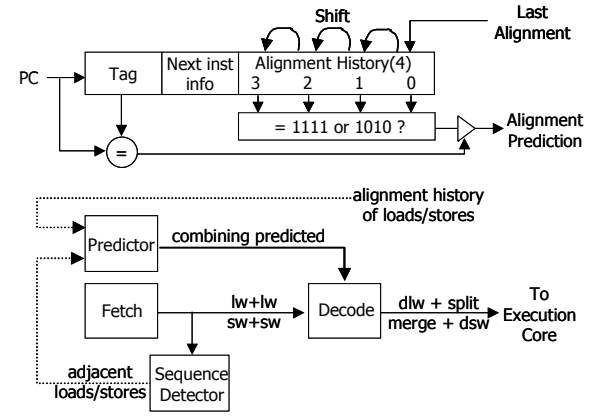


Figure 7. The structure of the combining predictor and the pipeline front-end.

Altivec extensions [20]) or, as in the case of hardware combining, require modifications to the load/store queue to hold wider values.

Detection / Prediction. Detecting combinable instructions that are not located near to each other in the dynamic instruction stream can be challenging. Furthermore, maintaining precise exceptions for store combining can be difficult in cases where an instruction between two combinable stores throws an exception. To simplify these issues, our mechanism is restricted to combining load sequences interrupted only by ALU instructions or uninterrupted store sequences. Finally, many architectures require naturally aligned memory accesses, and misaligned references can cause exceptions or soft traps. However, it is hard to statically decide whether the combined reference will be naturally aligned at decode time. We propose a structure called the *Combining Predictor* to predict the alignment of the combined references and direct the decoder to perform transformations. Its structure and the pipeline are illustrated in Figure 7. The sequence detector monitors the dynamic instruction stream and detects combinable references by examining offset fields that differ by 4 with the same base register and also ensuring the second load's base register is not overwritten by the first instruction. Although sub-word references (e.g. load-halfword and load-byte) could be combined into wider references, we did not consider such cases since they happen infrequently. The sequence detector can be easily implemented by using a

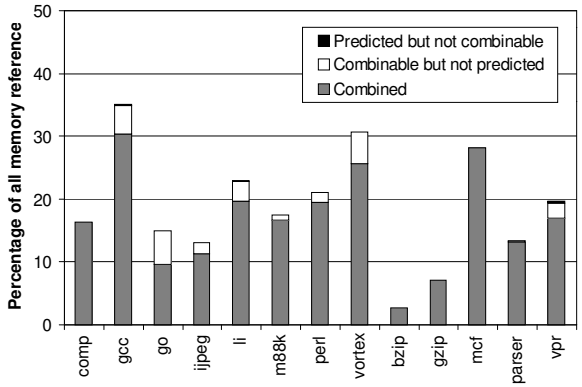


Figure 8. Percentage of combined memory instructions captured by a 1024-entry predictor.

few narrow adders and gates to examine offset fields and base registers, as presented in [10]. Because the sequence detector only adds a new entry to the combining predictor and it is located outside the decode path, the processor's critical path will not be affected by the latency for detections. When a combinable group is identified, a new predictor entry is allocated in the predictor.

A predictor entry records the last 4 alignment outcomes under the assumption that two instructions are combined. After several instances, references are combined based on bit patterns stored in the history fields if the pattern indicates that the next reference is likely to be double-word aligned. Table 3 summarizes the bit patterns and corresponding predictions.

Misprediction Recovery. When a combined reference tries an unaligned access to memory due to a misprediction, it is detected by the existing exception handling mechanism. The pipeline is drained and the original instruction are fetched again, and instructions are decoded without any combining transformations after the recovery.

In Figure 8, we show the coverage and accuracy of a double-word combining predictor with 1k direct-mapped entries. The percentages of combined instructions range from 2.67% of all memory references in *bzip* to a maximum of 30.4% in *gcc*. Our predictor captures roughly 80% of combinable references in all cases except for *go*, in which only 64% are captured. On the other hand, the misprediction rate due to misaligned memory accesses is extremely low (0.1% of all memory references in most benchmarks). Performance improvements from double and quad-word memory reference combining are discussed in Section 5.2, compared with optimistic hardware schemes [9][19] discussed in the previous section.

4.2. Silent store squashing

A silent store is an instruction which writes a value that exactly matches the value already stored at the memory location that is being written [3][5][7]. Because silent stores do not change the machine state, eliminating them is safe and has several advantages: reducing the pressure on cache write ports, reducing the pressure on store queues or other microarchitectural structures used to track pending

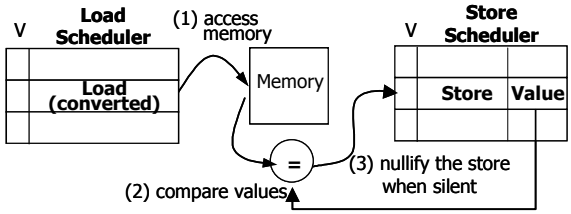


Figure 9. A possible in-core implementation for silent store squashing.

writes, reducing the need for store forwarding to dependent loads, and reducing both address and data bus traffic to lower levels in the memory hierarchy [3][5].

4.2.1. In-core implementations

The initial approach for eliminating silent stores as originally proposed in [5] requires converting each store into three implicit operations; a load, a comparison, and a conditional store that is initiated when the memory value and the new value to be written do not match (illustrated in Figure 9). As pointed out in [4], this simple approach has drawbacks when it is applied to every single store; it can place additional pressure on cache ports and other execution resources since unnecessary load and compare operations are performed even for non-silent stores. The next section introduces silence prediction, which can be used to reduce the frequency of silent store verifies.

An additional complication for store verification arises from the fact that many processors [16][18] have separate scheduling queues for loads and stores. Separate load/store schedulers imply that a store instruction must occupy both queues: the load queue for the store verify, and the store queue for the conditional store. Naturally, this will increase contention. Furthermore, multicycle scheduling latency for store ports may negate much of the benefit of squashing silent stores, since a successful store verify must complete releasing and reallocating the port before the store issue.

Additional techniques for reducing the cost of store verifies are discussed in [4]; we do not consider them here since they entail significant changes to the processor's load/store queues and/or data cache hierarchy. Our approach is able to achieve comparable performance benefits.

4.2.2. Silent Store Squashing via SD

SD silent store squashing is implemented by removing the store from the decoded instruction stream. Instead, a store is decoded into a load and a compare that verify its silence to guarantee correctness. Since verify operations are explicitly injected into the processor core, implicit load verifies and conditional stores do not adversely affect the scheduling logic. However, transforming all stores incurs recovery overhead when they are not silent; therefore, we need a prediction mechanism that filters out many of the non-silent stores.

Silence Prediction. Training a silence predictor is difficult because silence outcomes become available only when store verifies are being performed. Once the predictor state machine reaches a terminal state and hence the predictor

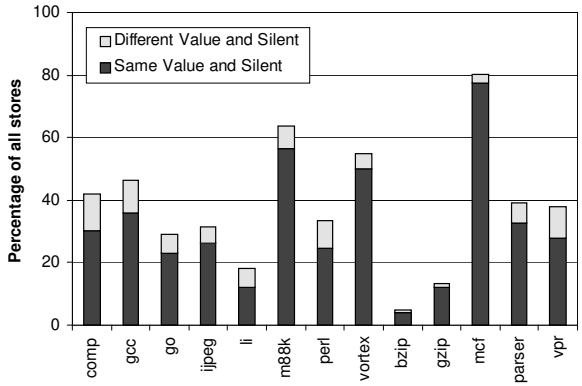


Figure 10. Percentage of silent stores categorized by the last store value.

determines that a store is not silent, no outcome history would be available to transition out of that state. Instead, our predictor mechanism exploits a silent store characteristic that was reported in [3]: static stores that consecutively write the same value are more likely to be silent than if they were storing different values. Figure 10 shows the likelihood of a store being silent as a function of whether a static store writes the same value as the last store value, regardless of effective memory addresses. We see that in most benchmarks over 70% of all silent stores can be captured by correlating with the last store value.

Our silence predictor is a PC-indexed direct-mapped table where each predictor entry has two saturating counters and a value history field as shown in Figure 11a. The value history field contains only lower 8 bits of the last store value since initial experiments determined that this was sufficient for providing reasonable prediction accuracy and significantly reduces the size of the predictor. Depending on whether the same value as the history is written, the confidence counter is increased or decreased. However, a store may not be silent even when the same value is stored repeatedly since the stores can be to a different address or intervening stores to the same location can alter the stored value. To reduce mispredictions due to such aliasing, our predictor mechanism has a dynamic threshold counter that changes according to the silence outcome history. We show the state diagram of the silence predictor in Figure 11b.

Silence Squashing Mechanism and Misprediction Recovery. After the same value is observed repeatedly by the predictor and the confidence reaches the threshold, the store is decoded into a store plus a load and a compare to check silence. If the store is silent, its threshold counter is decreased; otherwise the threshold is increased by a fixed penalty and the confidence counter is cleared, which makes it harder for the store to transition out of the 'No squash' state. When a store is predicted to be silent, only a load and a conditional trap are generated until a store verify fails. Figure 13 shows an example of SD silent store squashing. If a silence misprediction in 'Squash' state is detected by a trap operation, the mispredicted store is fetched and issued again without transformations after draining the pipeline.

In Figure 12, we show the performance of the silence

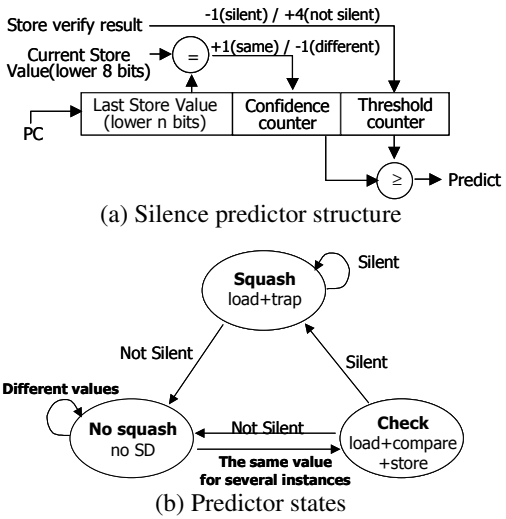


Figure 11. The structure of a silence predictor and its state diagram.

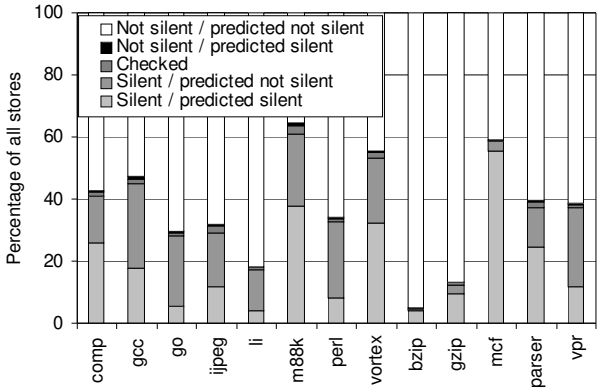


Figure 12. Percentage of silent stores captured by a predictor with 1k entries and 6-bit threshold counters.

predictor used in our simulations, with the configurations of 1024 direct-mapped entries, 8-bit last value fields, and 6-bit confidence and threshold counters (1k X (8+6+6 bits) = 2.5 kB). It correctly predicts (silent / predicted silent) 45.5% of all silent stores on average, which accounts for 5~55% of all dynamic stores. Compared with same-value-silent stores shown in Figure 10, some benchmarks such as *go*, show relatively low prediction rates since many non-silent stores also repeatedly write the same value and hence decrease the efficiency of our predictor. The misprediction rates (not silent / predicted silent) range from 0.02% in *li* to 1.7% in *m88ksim*. Performance via SD and in-core implementations will be discussed later in Section 5.3.

5. Microarchitectural Evaluations

5.1. Performance effects incurred by SD

Speculative decode can affect program executions in the following ways:

• *Replacement effect*: Original instructions are translated

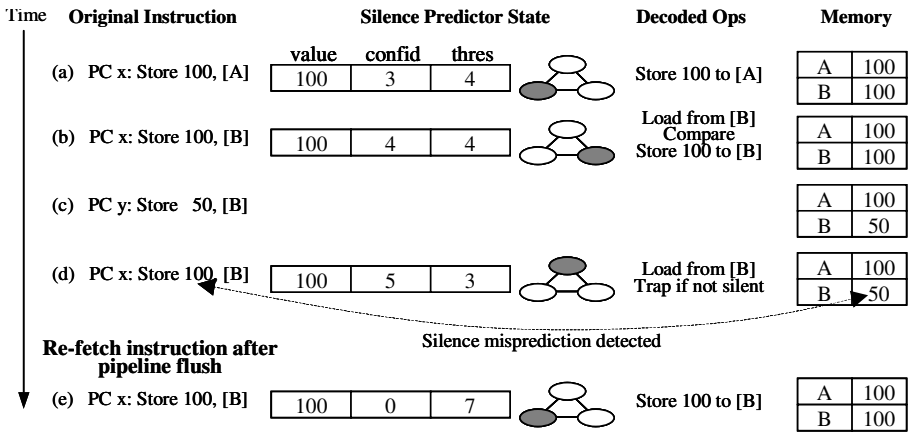


Figure 13. An example of silent store squashing via SD.

(a) The confidence counter is increased regardless of effective addresses if the store value matches the history in the predictor
(b) The confidence reaches the threshold and a load is issued to see if the store is silent. The threshold is decreased when it is silent.
(c) An intervening store changes the value of memory B.
(d) Only a load and a trap are issued when the confidence exceeds the threshold. The trap is triggered when values do not match.
(e) After the misprediction, the confidence counter is cleared and the threshold is increased by a penalty (4 in this example) and the store is fetched and issued again after draining the pipeline.

into semantics that interact differently with surrounding operations. For example, stores in silent store squashing are decoded as loads that do not create RAW or WAW memory dependences for subsequent instructions.

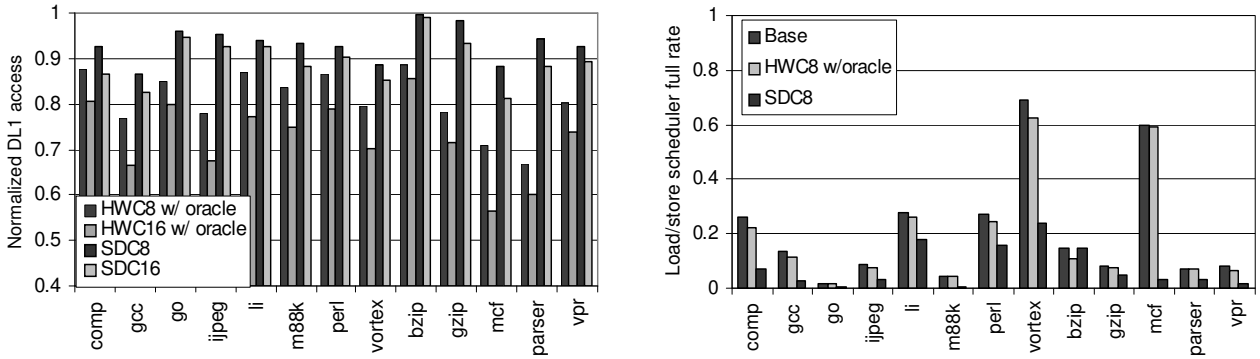
- *Bandwidth effect*: The total number of instructions executed changes due to SD. If an instruction expands into several operations, SD consumes more dispatch/issue/commit bandwidth, queue entries and functional units, which may negatively affect the performance. Conversely, SD may help to reduce bandwidth consumed by fewer operations.
- *Latency effect*: Extra operations in a depencence chain increase the instruction latency. E.g. piggyback loads get values after extract operations (`exthi` in Figure 6), which may increase their latency.

5.2. Evaluations of SD for memory reference combining

We evaluate SD double-word (SDC8) and quad-word (SDC16) combining, comparing with in-core implementations that perform double (HWC8) and quad-word (HWC16) combining together with store merging, similar to [9][19]. As discussed in Section 4.1.1, HWC is not easy to integrate into a pipeline with realistic scheduling latency. For purposes of comparison, we assume an optimistic oracle scheduler that identifies piggyback loads at schedule time before effective addresses are known.

Figure 14a shows normalized DL1 cache accesses reduced by HWC and SDC, with respect to the base machine. Since HWC is not restricted to adjacent instructions and combines across several references, it captures significantly more references and in turn reduces more cache accesses than SDC in all benchmarks (on average 19.3% (HWC8), 6.8% (SDC8), 27.5% (HWC16) and 10.4% (SDC16), respectively). Based on the cache accesses, one might expect that HWC would give much greater performance improvements than SDC. However, the improvements come from different sources: fewer memory instructions reduce cache accesses in SDC while HW combining achieves the reduction by buffering them. SDC gets benefits from the positive *bandwidth* (fewer memory instructions) and *replacement* (fewer address generations) *effects* in load/store schedulers and ALUs. On the other hand, HWC still performs dispatch, issue and commit of all memory instructions and experiences the same amount of resource contention as in the base machine, except that the improved throughput helps to resolve some contention. Figure 14b shows that there is less load/store scheduler contention in SDC in all cases except for *bzip*, in which SDC achieves a very low rate of combining.

We show the speedups achieved by SDC and HWC in Figure 15. In general, SDC achieved comparable speedups although HWC captures many more memory references than SDC. In *gcc*, SDC shows even better improvements. In some benchmarks SDC8 shows better performance than SDC16 because quad-word alignments are harder to predict than double-word alignments and result in higher misprediction rates. It is interesting that the contention reduction in SDC does not always correlate to better performance improvements. Besides the differences in the numbers of combined references, this is mostly because SDC transfers the contention from the load/store schedulers to the RUU, whose contention becomes another bottleneck (RUU full rates of $0.14 \rightarrow 0.41$ and $0.24 \rightarrow 0.77$ in *vortex* and *mcf*, respectively). In *mcf,* frequent cache misses cause the RUU to be filled up, which prevents speculatively decoded extract instructions from being committed. Another reason SDC does not always improve performance is the *latency effect* of extract operations. When piggyback loads are on the program critical path, performance is degraded compared to the base case in which loads are issued in parallel. If SDC were applied selectively depending on memory port contention or criticality of load latency, we would expect greater speedups



(a) Normalized DL1 accesses (b) Load /store scheduler full rates

Figure 14. Normalized DL1 accesses and load/store scheduler full rates in HW and SD memory reference combining.

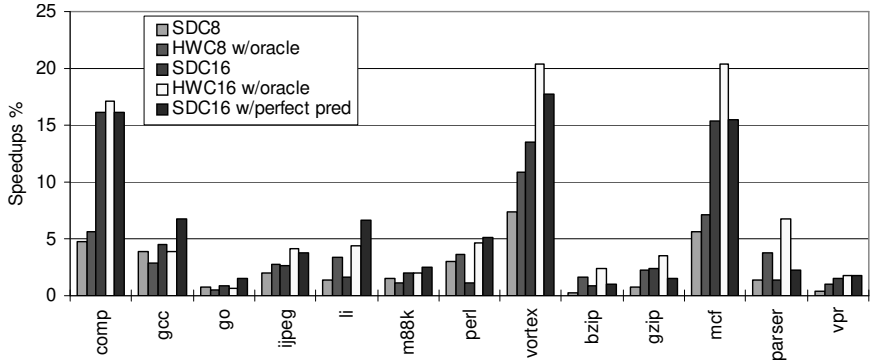


Figure 15. Speedups from HW and SD memory reference combining.

from it. The rightmost bars in Figure 15 are the potential SDC16 speedups from a perfect combining predictor and show that SDC with a perfect predictor outperforms HWC with the oracle scheduler in some benchmarks.

In summary, memory reference combining via SD provides comparable or even better performance than the impractical in-core implementation because SD pre-schedules this optimization and eliminates contention more effectively.

5.3. Evaluations of SD for silent store squashing

To evaluate SD for silent store squashing (SDSSS), it is compared with the predictor-based in-core silent store squashing (HWSSS), derived from [5]. A load verify is performed only on predicted stores in both cases. This HWSSS has several advantages over SDSSS. First, HWSSS does not need any recovery such as draining the pipeline in SD from mispredictions, since the store is conditional depending on the store verify. Second, it does not require the 'check' state of the silence predictor because the purpose of the state was to reduce the misprediction penalty. Third, the dedicated compare logic reduces contention in the ALUs. We discussed the details of HWSSS, and potential difficulties in Section 4.2.1. The silence predictor and SDSSS were described in Section 4.2.2.

Detecting silent stores may improve memory disambiguation, allowing later loads to bypass earlier stores with unresolved effective addresses [11]. Delaying the later loads after unresolved silent stores to avoid memory dependence violations is unnecessary because silent stores do not change the values of the memory location. Improved memory disambiguation is achieved by the *replacement effect* of SDSSS since predicted silent stores are converted into loads that do not block subsequent memory accesses. On the other hand, HWSSS still dispatches silent stores and requires extra logic to obtain this benefit [11]. Figure 16 shows the average extra cycles needed for a load instruction to wait until earlier store addresses are resolved. We measure a 14% reduction in store-to-load block cycles in SDSSS compared to those of the base machine. It is expected that the potential performance improvements would be greater on register-starved ISAs such as Intel x86, on which the average performance improvement is reported as 4%, ranging from 0.2% to 14.7% according to the initial study by Yoaz et. al.[11].

Figure 17 reports the performance improvements from HWSSS and SDSSS when they use both a silence predictor previously discussed and a perfect silence predictor. Speedups are calculated based on the total execution time since SD changes the total number of committed instructions and IPC numbers are no longer comparable. Performance degrades in SDSSS marginally on half of the benchmarks where HWSSS also shows minor speedups or slowdowns. This is primarily because SDSSS suffers from silence mispredictions. Especially in *m88ksim* the performance is degraded noticeably due to a high misprediction

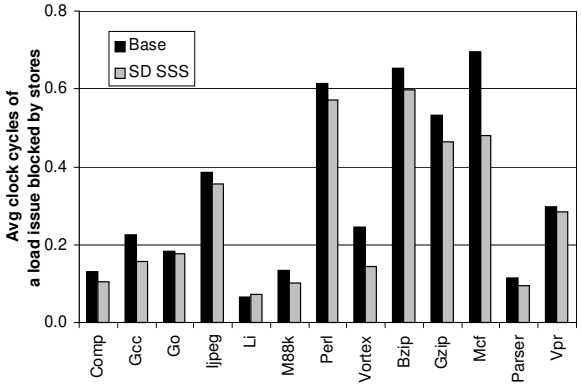


Figure 16. Improved memory disambiguation via SD silent store squashing.

rate. The negative *bandwidth effect* is another source of SDSSS slowdowns since extra silence verify operations in check and squash states consume ALU and cache port bandwidth.

On the other hand, significant performance improvements are observed in *compress, vortex* and *mcf* (up to 13.8%, 18.1% and 19.7%, respectively). The perfect predictor in *mcf* achieves lower speedup than our silence predictor due to a second-order LRU effect in the L2 cache: since silent store squashing avoids setting the dirty bit in the L1 cache, lines are evicted silently and do not update the L2 LRU, leading to additional L2 misses. In these benchmarks, SDSSS outperforms HWSSS with a perfect predictor because SDSSS benefits from improved memory disambiguation and HWSSS does not reduce contention in the store scheduler.

5.4. Performance sensitivity to pipeline depth

As discussed in Section 2.3, SD should be evaluated critically if the processor does not have a translation stage or SD requires extra decode latency. Figure 18 shows the performance sensitivity to the decode stage depth when both double-word memory reference combining (SDC8) and silent store squashing (SDSSS) are performed. In this graph, the execution time is normalized to the base machine with 1 decode stage. Although performance degrades in all cases as the decode stage pipeline gets deeper, 8 benchmarks (*compress, gcc, ijpeg, vortex, bzip, gzip, mcf* and *parser*) still perform as well or better than the base machine with no extra stage even when SD requires 1 extra pipeline stage. All benchmarks except for *m88ksim* show speedups compared with each base case. To better understand the effect of pipeline depth on SD performance, we present detailed sensitivity results on three benchmarks that achieve significant speedups from our optimizations in both SD and HW.

In Figure 19a, we show performance sensitivity of base, HW and SD to the decode pipeline depth. The front-end pipeline depth affects overall performance by increasing the time spent on recovery from either branch or speculative optimization mispredictions (alignment and silence in our study). Although the penalty for a SD misprediction is significant and HW does not suffer from it, the relative performance degradation in SD is surprisingly slight, mainly because our SD predictors are highly accurate.

On the other hand, additional pipeline stages between schedule and writeback (EX pipeline) have greater impact on performance than additional front-end stages. Besides a longer misprediction penalty and more mis-scheduled instructions under load shadow [13], a longer EX pipeline increases contention in out-of-order execution queues due to increased occupancy. HW does not reduce or even increases contention while SD reduces queue occupancy in the memory schedulers and hence HW performance is noticeably more degraded than SD as the EX pipeline gets longer, as presented in Figure 19b. In summary, SD may provide benefits in an even longer pipeline where HW would fail to do so since SD is less sensitive than HW to this portion of the pipeline.

6. Related Work

Jacobson et. al. [1] and Friendly et. al. [8] suggested transforming instructions based on peephole optimizations during a trace cache construction to implement better instruction scheduling, constant propagation, instruction collapsing and etc. Chou and Shen [2] proposed the instruction path co-processor, which is a programmable internal processor that operates on instructions of the core processor to transform them into a more efficient stream, and showed the performance gain from similar optimiza-

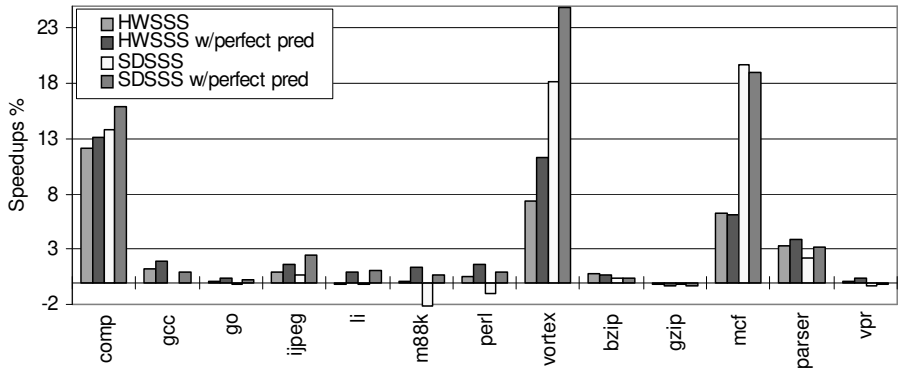


Figure 17. Speedups from silent store squashing via HW and SD.

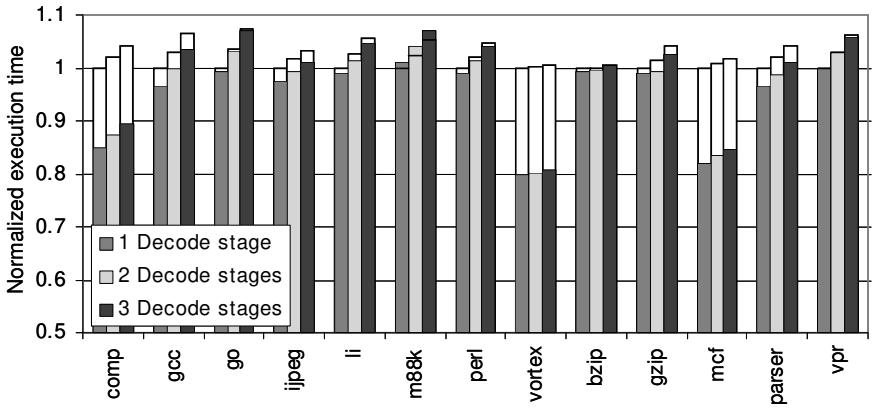


Figure 18. SD performance varying the decode pipeline depth. Stacked white bars show base execution time when no optimization is performed (slowdowns in *m88ksim* indicated as lines).

tions. Prior work has focused primarily on safe code optimizations in the presence of hardware mechanisms such as predication or trace cache. Rather, SD may be efficiently implemented through those transformation mechanisms. The most significant difference between our approach and these prior experiments is that we relax the *safety* constraint. That is, by employing speculative transformations that are not conservatively guaranteed to be safe with respect to program semantics, we expose greater opportunity for performance enhancement and simpler implementations in a manner compatible with speculative scheduling. Borch et. al. [13] studied the effect of critical architectural loops in speculative scheduling and proposed a hierarchical register file design to shorten the EX pipeline. On the other hand, SD reduces contentions in the EX pipeline by pre-scheduling optimizations and hence it is less sensitive to the longer EX pipeline than in-core implementations.

6.1. Memory reference combining

Wilson et. al. [9] proposed a comprehensive evaluation of several techniques that subsequent accesses to the same line are satisfied by a single memory access. This approach discusses on improving the effective memory bandwidth between the processor core and the data cache with a restricted number of cache ports. It assumes a generic out-of-order pipeline with atomic wakeup/select and does not consider the problems that may occur in speculative scheduling. Lopez et. al. suggested a hardware mechanism to compact two load/stores into a single wide reference with help of the compiler [10]. This work is conceptually similar to the approach via SD but the latency needed to detect references, memory alignment and merging and splitting values into/from a wider instruction were not discussed.

6.2. Silent store squashing

Molina et. al. [7] and Lepak and Lipasti [5] showed there is a significant amount of redundancies in store instructions. [7] proposed to put a simple comparator to reduce unnecessary cache accesses due to store instructions. The initial approach to eliminate silent stores in [5] is converting all stores into store verify operations. [4] proposed several lower-cost store verify approaches: one is verifying stores using only idle load ports and the other is exploiting temporal and spatial locality in the load/store queue, issuing no extra load. Considering the scheduling distance and contentions in separate load/store schedulers, we demonstrated that SD is a better approach to implement this optimization. Yoaz et. al. [11] proposed the path information-based silence predictor. They also suggested other possible applications of detecting silent stores to exploit dead instructions and enhance memory disambiguation. Although their silence prediction shows very high coverage and accuracy, their work did not discuss the training cost incurred by verifying silence for every store.

7. Conclusions

We make five major contributions in this work. First, we describe the difficulties in implementing in-core optimizations in a realistic processor pipeline. Second, we introduce speculative decode as a feasible mechanism for implementing late, run-time optimizations. Third, we present two novel compact predictor designs: memory reference combining and silence predictors with very low implementation cost and high coverage and accuracy. Fourth, we demonstrate that memory reference combining and silent store squashing implemented via speculative decode can perform comparably or better than impractical in-core implementations. Fifth, we study the effect of pipeline depth on the overall performance and show that optimizations via speculative decode are less sensitive to a longer execution pipeline than in-core implementations.

8. Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437, and generous financial support and equipment donations from IBM and Intel. We would also like to thank the anonymous reviewers for their many helpful comments.

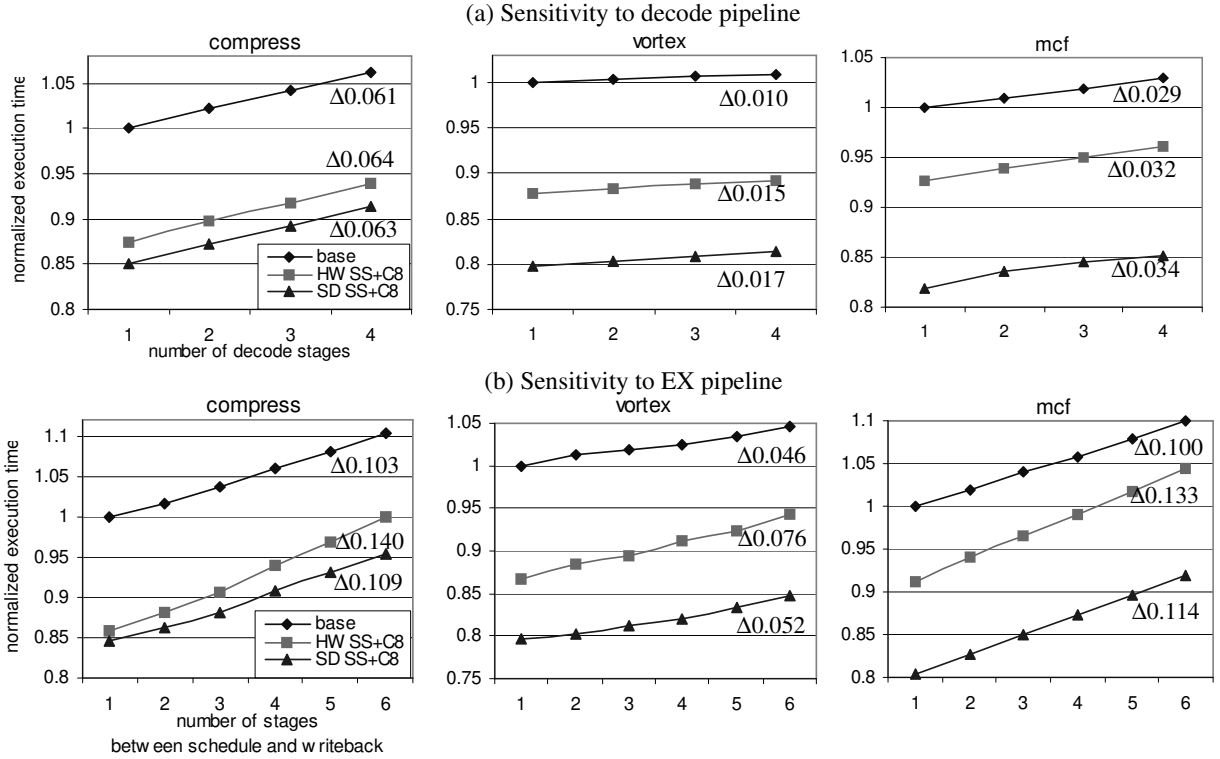


Figure 19. Performance sensitivity to the pipeline depth. The number denoted with Δ shows the normalized execution time difference on each case.

9. References

[1] Q. Jacobson and J. E. Smith, *Instruction Pre-Processing in Trace Processors*, in Proc. of 5th International Symposium on High Performance Computer Architecture, 1999.

[2] Y. Chou and J. P. Shen, *Instruction Path Coprocessors*, in Proc. of 27th International Symposium on Computer Architecture, June 2000.

[3] G. B. Bell, K. M. Lepak and M. H. Lipasti, *Characterization of Silent Stores*, in Proc. of International Conference on Parallel Architectures and Compilation Techniques, October 2000.

[4] K. M. Lepak and M. H. Lipasti, *Silent Stores for Free*, in Proc. of 33rd International Symposium on Microarchitecture, December 2000.

[5] K. M. Lepak and M. H. Lipasti, *On the Value Locality of Store Instructions*, in Proc. of 27th International Symposium on Computer Architecture, June 2000.

[6] 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.

[7] C. Molina, A. Gonzalez and J. Tubella, *Reducing Memory Traffic Via Redundant Store Instructions*, in Proc. of Conference on High Performance Computing and Networking, 1999.

[8] D. Friendly, S. Patel and Y. Patt, *Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors*, in Proc. of 31st International Symposium on Microarchitecture, December 1998.

[9] K. M. Wilson, K. Olukotun and M. Rosenblum, *Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors*, in Proc. of 29th international symposium on Microarchitecture, 1996.

[10] D. Lopez, M. Valero, J. Llosa and E. Ayguade, *Increasing Memory Bandwidth with Wide Buses: Compiler, Hardware and Performance trade-offs*, in Proc. of International Conference on Supercomputing, 1997.

[11] A. Yoaz, R. Ronen, R. S. Chappell and Y. Almog, *Silence is Golden?*, in work-in-progress workshop of 7th High-Performance Computer Architecture, January 2001.

[12] R. Hilgendorp and W. Sauer, *Instruction Translation for an Experimental S/390 Processor,* in Workshop on Binary Translation in Parallel Architectures and Compilation Techniques, October 2000.

[13] E. Borch, E. Tune, S. Manne and J. Emer, *Loose Loops Sink Chips*, in Proc. of 8th International Symposium on High Performance Computer Architecture, 2002.

[14] Compaq Computer Corporation, *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[15] B. Calder, D. Grunwald and J. Emer, *Predictive Sequential Associative Cache*, in Proc. of 2nd International Symposium on High Performance Computer Architecture, 1996.

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

[17] A. Moshovos and G. S. Sohi, *Speculative Memory Cloaking and Bypassing*, in Proc. of 32nd International Symposium on Microarchitecture, December 1999.

[18] J. M. Tendler et. al., *POWER4 System Microarchitecture*, IBM technical white paper, October 2001.

[19] D. A. Patterson and J. L. Hennessy, *Computer Architecture: a Quantitative Approach*, 2nd ed., p. 382, Morgan Kaufmann, 1996.

[20] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scale, *AltiVec Extension to PowerPC Accelerates Media Processing*, IEEE Micro, vol. 20, no. 2, pp. 85-95, 2000.

[21] R. M. Tomasulo, *An Efficient Algorithm for Exploiting Multiple Arithmetic Units*, IBM Journal, Vol. 11, pp. 25-33, January 1967.