Performance and Power via Speculative Decode

Ilhyun Kim

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

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

May 1, 2001

* This material is based upon work supported by National Science Foundation under Grant No. 0073440. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

1 .0 Introduction and Motivation

Indisputable empirical evidence supports our claim that a processor core microarchitecture tends to survive largely unchanged across multiple implementations and process technology generations. Two recent examples--the Intel Pentium and Pentium Pro core designs--both survived nearly intact for five or more years and were realized in multiple process technologies [27]. Similarly, core designs from IBM, Sun SPARC, Hewlett-Packard, and Alpha have proliferated across several implementations and process technology generations. This *core reuse* effect is caused primarily by the time and extreme expense associated with the design and verification of a brand-new processor microarchitecture, and we fully expect it to continue for the foreseeable future. While process technology generations turn around every one or two years, the design cycle for a new core can take up to five years [29]. This mismatch between design cycles and technology generations, combined with the incessant marketplace demand for increased performance, leads design teams to generate frequent iterations or remaps of the same core design to new process technologies to enable higher performance, smaller die size, and lower power consumption.

At the same time, both academic and industry researchers continue to propose increasingly sophisticated microarchitectural techniques that enhance performance or reduce power or energy consumption. Unfortunately, these techniques often appear to require significant changes to existing cores, inevitably delaying their adoption until the next major core redesign that enables incorporation of such changes. In this paper, we demonstrate that it is possible and in fact desirable to incorporate new microarchitectural techniques into existing core designs with minimal perturbation of the core itself. Our approach is enabled by the fact that there is often a translation layer that exists to bridge the gap between the user-visible architected instruction set (*U-ISA*) and a realizable implementation instruction set (*I-ISA*). This gap exists because direct implementation of complex instructions is deemed infeasible or unprofitable by the core designers. There is a long and rich history of varying approaches for this translation layer, ranging from trap-based translators where unimplemented instructions are emulated in the operating system's invalid instruction exception handler (e.g. early MicroVAX implementations and more recently, the PowerPC 604, which dropped direct support for certain complex POWER instructions [20]); to microcoded emulation routines stored in an on-chip lookup table (e.g. Pentium Pro and it's derivatives [21]); to software-based binary translation approaches (e.g. FX!32[22], Daisy[23], BOA[25] and Crusoe's code morpher[24]). A recent example of a design that incorporates a microcode translation layer to simplify the core implementation is the Power4 design from IBM, which abandons direct support for the complex storage instructions in the PowerPC instruction set, instead emulating them with microcoded streams of primitive load and store operations [16].

The prior work most closely related to ours has focused primarily on safe code transformations based on peephole optimizations within a limited scope. Specifically, the instruction path co-processor work [3] as well as earlier trace cache peephole optimizations[2][26] focused specifically on safe scheduling and code transformation optimizations within the scope of a single trace cache entry. In contrast, our experimental framework does not assume the existence of a trace cache. However, the transformations we propose could just as easily be implemented as off-line operations on trace cache entries. We avoid this approach because while it is an obvious implementation choice for core designs that already implement a trace or decode cache (e.g. Pentium-IV [13]), we did not want to unnecessarily constrain our study to this specific implementation choice, nor did we want to obscure our results with performance and power consumption artifacts that would be induced by this choice. We plan to study these issues in future work. The most significant difference between our approach and these prior experiments--besides the fact the we study a different set of transformations--is that we relax the *safety* constraint. That is, by employing speculative transformations (within our *speculative decode* framework) that are not conservatively guaranteed to be safe with respect to program semantics, we expose greater opportunity for performance enhancement and power savings. Of course, we still have to guarantee correct semantics by checking for correctness violations and recovering from incorrect speculation.

However, we find that for the two speculative techniques we study, the overhead of misspeculation recovery is negligible in nearly all cases. We published an initial workshop paper that examined speculative decode in the context of binary translation between two incompatible instruction sets[1]. In this study we found that two incompatibilities between the PowerPC and PISA instruction sets--alignment requirements and support for variable-length load/store instructions--can be effectively addressed with the *speculative decode* approach.

In this paper we extend the speculative decode approach to encompass two microarchitectural techniques that address inefficiencies in the handling of memory reference instructions. First, we study the opportunities for load and store reference combining that 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 64-bit data paths in support of instruction set extensions that have now been added to all major general-purpose instruction sets (the last two 32-bit holdouts, IBM’s S/390 and Intel x86 or IA-32, both recently announced 64-bit extensions and implementations). Despite these 64-bit extensions, it is widely expected that the vast majority of user-mode and even kernel code will continue to execute in 32-bit mode for years to come, since very few user applications require the massive address space provided by the 64-bit extensions. We find that significant performance improvements and reductions in energy consumption and energy-delay product are possible with this very simple speculative transformation. Further details of this technique are presented in Section 3.1.

The second technique we study is speculative decode in support of store verify operations for squashing of silent store instructions [7][4][5]. In this study, we employ a novel value-history-based predictor that allows us to identify a significant fraction of all silent stores in a program’s execution while minimizing the overhead associated with detecting and verifying silent stores. In the naive original proposal[7], all stores were converted into store verify operations consisting of a load, compare, and conditional store. Whenever the store was not silent, the load and compare induced additional resource contention and activity into the processor core. A subsequent study[5] showed that microarchitectural modifications can significantly reduce this overhead. However, that study assumed that significant core changes can be made in support of efficient store verification. In contrast, this work demonstrates that most of the benefits of core-compatible store verification are achievable with much less additional resource contention and activity by employing a novel *silence predictor*, resulting in performance benefits and reductions in energy consumption. Further details of this technique are presented in Section 3.2.

Analysis of performance and power consumption for these techniques separately and in combination are presented in Section 4. Finally, Section 5 summarizes our findings and discusses opportunities for future work.

2 .0 The Concept of Speculative Decode

Speculative decode is a new technique to exploit runtime attributes by speculatively and optimistically decoding static instructions into an advantageous sequence rather than non-speculatively decoding it into a sequence of instructions that will work correctly in all cases. Figure 1 illustrates the basic concept of speculative decode. The predictor accesses the non-architected entities of the core processor and gathers necessary information for speculation. The predictor is accessed in parallel with instruction fetch by using PC value to overlap the delay of accessing the extra table. If a certain opportunity for optimization is predicted based on the information collected by the predictor, the decode logic replaces an instruction or a series of instructions with a different instruction stream which includes working instructions as well as verification code. It may be argued that increasing the complexity of the decode logic to support this translation may be more difficult than it appears to a naive observer. However, since translation between architected ISA and implementation ISA occurs in many processors, we believe that the current generation decoder will be able to incorporate speculative decode without significantly affecting the processor cycle time. For example, Intel P6[14], AMD K6[15], and IBM Power4[16] include a translation layer to maintain compati-

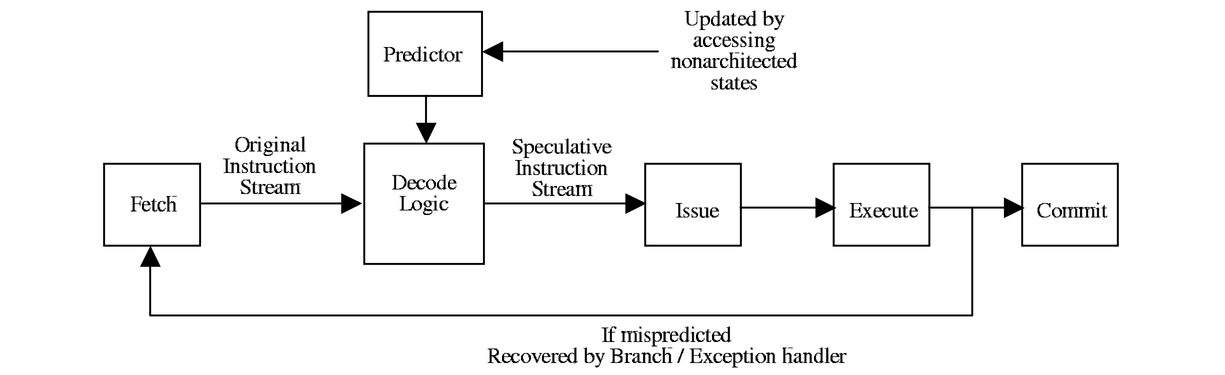

Predictor
Updated by accessing nonarchitected states
Original Instruction Stream
Speculative Instruction Stream
Fetch
Decode Logic
Issue
Execute
Commit
If mispredicted
Recovered by Branch / Exception handler

Figure 1. Basic concept of speculative decode integrated in a generic pipeline.

```
load r1, 36[r29]
add r3, r1, r5
```

```
load r1, 36[r29]
p2 = predicted value
bne r1, p2, softexception
add r3, p2, r5
```

Figure 2. Example code of load value prediction implemented by speculative decode. The predicted value is assigned to a non-architected register and a branch instruction is used to guarantee correctness.

bility between the architected ISA and the implementation ISA. Alternatively, it is possible that an additional functional block, such as an instruction path coprocessor[3], can be used to implement speculative decode. Furthermore, the transformations applied to implement speculative decode can be removed from the critical path by performing them off-line on a translated stream that is stored in a trace cache-like structure [28], as is suggested in prior work [2][3].

Speculative decode enables us to reuse the existing architectural resources to implement new features via translating instructions. Practically speaking, many of the speculative techniques proposed in the literature can be implemented by a series of instructions based on speculative decode. An example implementation of load value prediction[17] with speculative decode is shown in Figure 2.

In this example, a non-architected register p2 is used to hold the predicted outcome of the load instruction. In the out-of-order processor, instructions dependent on the load instruction are executed with the predicted value assigned by the move instruction and are prevented from committing until the load instruction is executed and the real value and the predicted value are compared by the branch instruction. If the prediction is correct, the branch instruction is executed transparently and the following instructions can be committed. If the values do not match, the branch misprediction handler is invoked by the branch instruction and the correct machine state can be maintained by draining the pipeline and invalidating speculative instructions.

One would expect that performance gains from speculative decode will be susceptible to misprediction frequency because the misprediction penalty is the same as the branch misprediction penalty. Since draining the pipeline is expensive, a very accurate prediction or confidence mechanism should be devised for a speculative decode mechanism if we do not have a less drastic selective invalidation and recovery mechanism. In Section 4.5, we will show that the effect of speculative decode mispredictions is not as severe as one might expect.

As well as performance benefit from speculation, speculative decode can be used to reduce power consumption. Although the logical semantics of the program are maintained, speculative decode can alter the way the program is executed by replacing a power-consuming instruction with another instruction that consumes less power(i.e. replacing a memory instruction with an ALU instruction). We will also show the power and energy reduction due to speculative decode in Section 4.6.

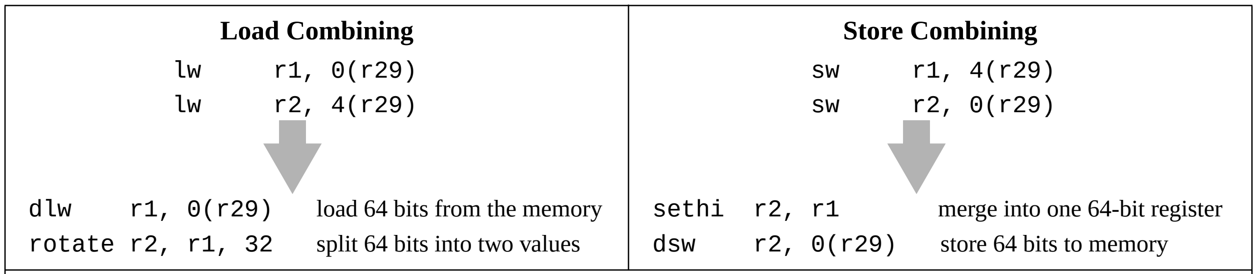
3 .0 Applications of Speculative Decode

In this section, we discuss two techniques based on speculative decode: load/store combining and silent store squashing [7]. These techniques focuses on reducing memory accesses by replacing load/store instructions with a different series of instructions to enhance performance and lower power consumption.

3.1 Load/Store Combining via Speculative Decode

The first speculative decode technique we evaluate is load/store reference combining. Here, we study the opportunities for converting 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 64-bit data paths in support of instruction set extensions that have now been added to all major general-purpose instruction sets, including Sun SPARC, HP PA-RISC, SGI MIPS, PowerPC, and even the last two 32-bit holdouts, IBM's S/390 and Intel x86 or IA-32, both of which recently announced 64-bit extensions and implementations (in the latter case, this announcement was made by AMD as part of the Sledgehammer project). Despite these 64-bit extensions, it is widely expected that the vast majority of user-mode and even kernel code will continue to execute in a backward-compatible 32-bit mode for years to come, since very few user applications require the massive address space provided by the 64-bit extensions. Of course, code executing in 32-bit mode will fail to exploit the wider data transfer paths and physical registers that must be present for efficient execution in 64-bit mode.

Load/store combining with speculative decode enables the use of these 64-bit resources. In load/store combining, two loads/stores that access consecutive memory locations can be merged into one wider memory access. We show possible examples of original and speculatively decoded code sequences in Figure 3.



Load Combining

```
lw     r1, 0(r29)
lw     r2, 4(r29)
```

```
dlw    r1, 0(r29)      load 64 bits from the memory
rotate r2, r1, 32      split 64 bits into two values
```

Store Combining

```
sw     r1, 4(r29)
sw     r2, 0(r29)
```

```
sethi  r2, r1          merge into one 64-bit register
dsw    r2, 0(r29)      store 64 bits to memory
```

Figure 3. Examples of speculatively decoded instruction sequences for load and store combining.

In the load combining example, we can statically detect two combinable word-size load instructions by examining offset fields that differ by 4 and checking if they have the same base register. The rotate instruction is inserted after the double load word instruction to put the higher 32-bit value to the lower order bits in the target register r2. The decoded sequence can vary depending on the instruction set architecture. For example, since the higher 32 bits loaded in the register are ignored when a 64-bit implementation of PowerPC architecture[10] is running in 32-bit mode, the sequence shown in Figure 3 works as illustrated as long as the 64-bit rotate instruction is allowed to execute. In the MIPS architecture[18], a double word load instruction puts a 64-bit values into two logically adjacent registers so the decoder logic may manipulate the rename table to put values into the arbitrary target registers without any ALU operation to extract the high word value. To evaluate the effectiveness of load/store combining that matches the more general case, we pessimistically assume that one double word load instruction and one ALU operation to split a value to two separate registers are needed for load combining. Store combining can be accomplished in an analogous manner. One ALU instruction (sethi in the example) to merge two 32-bit values into a 64-bit register and one double word store instruction are needed for store combining.

There are several problems with speculative decode with respect to reckless load/store combining. First, it is not feasible to detect combinable pairs in the instruction stream when two memory operations are not located near to each other in the dynamic instruction stream. Second, maintaining precise exceptions for store combining can be difficult in cases where an instruction between two combinable stores throws an exception due to, e.g., overflow. To avoid these problems, our mechanism is restricted to two adjacent, word-size load/store instructions with constant offsets. Because there will not be any change in control and data dependency between adjacent two memory instructions, we can statically determine combinable pairs safely.

Some architectures require memory instructions to maintain aligned access to memory locations; however, this is not true for the PowerPC architecture[10]. The most common restriction in memory access alignment is natural alignment, where an effective address for a memory location should be a multiple of data size. When two word-aligned memory instructions are combined into a double-word instruction, it is hard to decide statically whether the combined instruction will be double word-aligned because we do not know the effective address until the sum of base register value and offset value has been computed.

We propose a structure called the *Combining Predictor* to predict the alignment of the combined load/store instructions and direct the front end of the pipeline to perform load/store combining. This process is illustrated in Figure 4 and Figure 5.

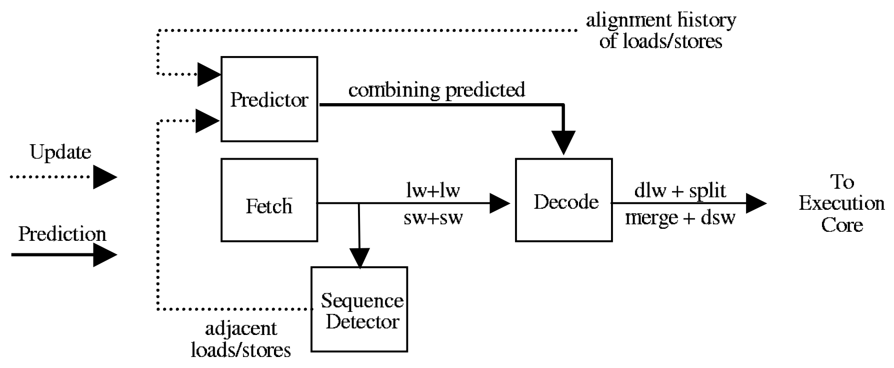


Figure 4. The frontend pipeline with a load/store combining mechanism. Two word-size adjacent memory instructions with the same base register and offset difference of 4 are decoded into a ALU for word splitting/merging and a doubleword-size memory instruction.

The sequence detector monitors the sequence of instructions fetched from the instruction cache and examines two adjacent, word-size load/store instructions with constant offsets. If offset fields differ by four, two instructions use the same base register, and also the second base register is not overwritten by the first instructions, it is guaranteed that those two adjacent memory instructions will access the consecutive memory locations. The sequence detector can be easily implemented by using a few narrow adders and gates to examine offset fields and base registers. Because the purpose of the sequence detector is to add a new entry to the combining predictor, it can be located outside the critical decode path. Therefore the processor's critical path will not be affected by the latency needed to detect sequences. Whenever a combinable pair is identified, a new predictor entry corresponding to the preceding memory instruction is added to the table.

A predictor entry shows a relationship between a memory operation at the current PC and an adjacent memory operation at the next PC. The alignment history field consists of 4 bits where the last 4 alignment histories are

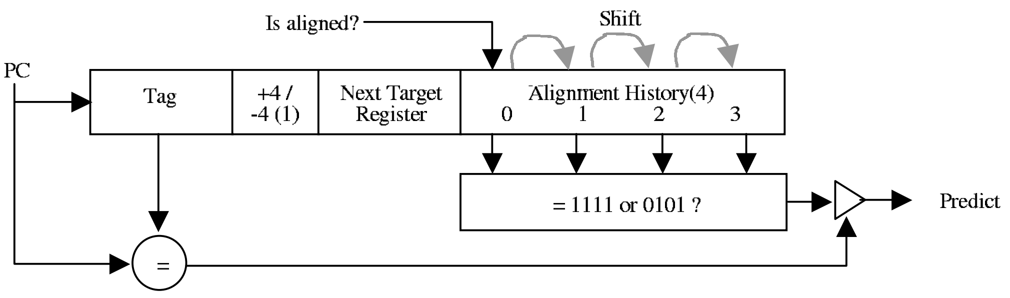


Figure 5. The structure of load/store combining predictor. A prediction is made simply by checking history bits to see if they match certain patterns.

stored under the assumption that adjacent loads/stores are combined. After predictor entries are updated for several instances by the effective addresses, predictions are made based on bit patterns stored in the history fields. When the pattern is 1111(aligned four times in a row), a combining operation is predicted since the next effective address is also likely to be double-word-aligned. Similarly, a 1010 bit pattern tells us that the base register value is increasing or decreasing by a word-aligned stride so that a pair of instructions can be combined on every other instance. Table 1 summarizes bit patterns and corresponding predictions.

If the preceding instruction is located at a fetch boundary, e.g., it is the fourth instruction in a four-wide machine, the load/store pair cannot be combined until we know the offset and target register identifier for the next instruction. Naively, we would delay until this instruction is fetched in the next clock cycle because we do not have sufficient information to generate speculative instructions. In order to avoid this delay, the predictor entry has additional fields of +4/-4 and a target register identifier which describe the next instruction. Because the prediction is made by comparing only bit patterns, while examining effective address to determine alignment and updating the predictor are done on the back end of the pipeline, it is unlikely that our predictor mechanism affects the processor critical path. Furthermore, we expect that the availability of decoded instruction caches(e.g. trace caches in Pentium-IV[13]) would allow speculative decode decisions to be moved completely off line.

Table 1: Confidence mechanism history bit patterns and their corresponding predictions

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

When a combined double-word reference tries an unaligned access to memory, it is automatically detected by the load/store queue, a soft exception is thrown, the pipeline is drained, and the faulting instruction is fetched again by the existing branch misprediction mechanism.

Figure 6 shows the shows the percent of combinable memory instructions for the SpecInt95 and SpecInt2000 benchmarks which are compiled by the gcc-pisa compiler with maximum optimization. The number of combinable memory instructions are quite significant even though we detect only adjacent instructions. In *gcc*, *vortex* and *mcf*, 28% ~ 35% of all memory references can be combined, which can lead to significant reductions in DL1

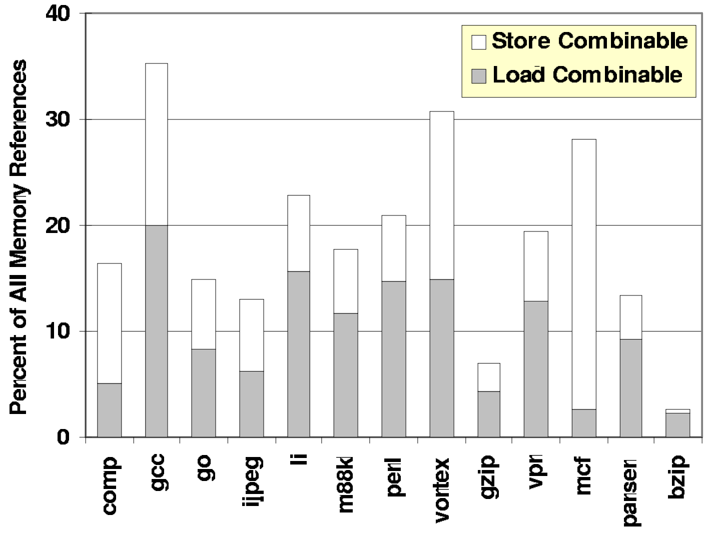


Figure 6. Percent of combinable loads/stores. By detecting two adjacent word-size loads/stores which access doubleword-aligned consecutive memory locations, on average 12% of memory instructions can be combined.

cache accesses. On average, 12% of memory instructions can be combined across the benchmarks we tested.

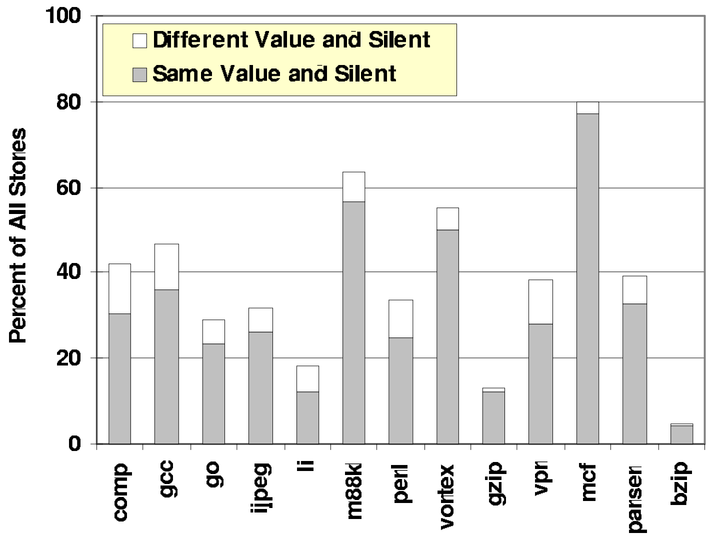


Figure 7. Percentage of silent stores categorized by the last store value. Grey bars show the silent stores that write a same value as the last time it was executed. White bars shows the silent stores that write a different value from the last time. In most cases, over 70% of silent stores write the same value as before.

3.2 Silent Store Squashing via Speculative Decode

The second speculative decode technique we examine is silent store squashing. A silent store is a instruction which writes values that exactly match the values that are already stored at the memory address that is being writ-

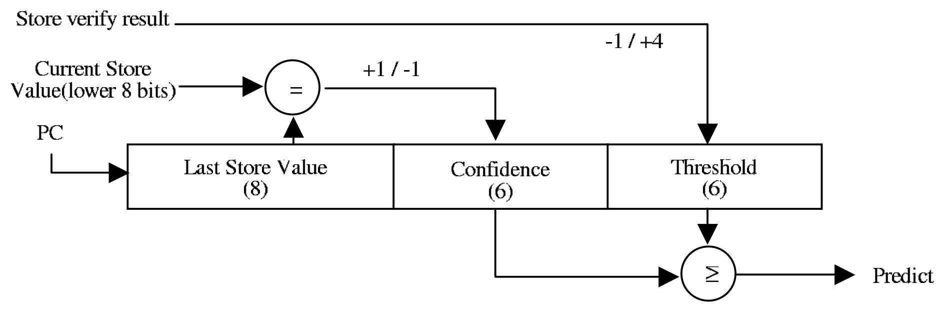


Figure 8. The structure of a silence predictor. A prediction is made by just comparing the confidence and threshold fields.

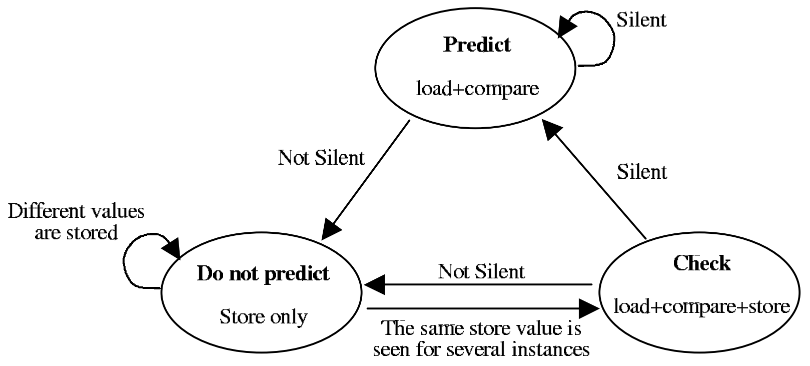


Figure 9. Silence predictor state transition diagram. Silent store squashing is predicted when confidence exceeds threshold.

ten[7][4][19]. 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 that are used to track pending writes, reducing the need for store forwarding to dependent loads, and reducing both address and data bus traffic to lower level in the memory hierarchy[7][4]. These benefits can lead to both performance improvement as well as reduced power and energy consumption.

The initial approach for eliminating silent stores as originally proposed in [7] requires converting each store into three 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. As pointed out in [5], this simplistic approach has several drawbacks when it is applied to every single store; it can place additional pressure on cache ports and other execution resources whenever a store is not silent since both the load and compare are still executed. [5] discusses other lower-cost approaches for verifying stores. However, all of these require non-trivial changes in the processor core. If we wish to avoid core changes and perform store verifies using existing resources, we need a prediction mechanism that filters out the non-silent stores among all stores in order to implement silent store squashing.

The biggest challenge for predicting silent stores is that updates to silence predictor history cannot be made based solely on the history of store outcomes (analogous to updates to branch prediction history with branch outcomes). In the case of a silence predictor, the silence outcomes are known only when the store verify is performed. When the confidence level is low and store squashing is not predicted, we do not know whether a store instruction is

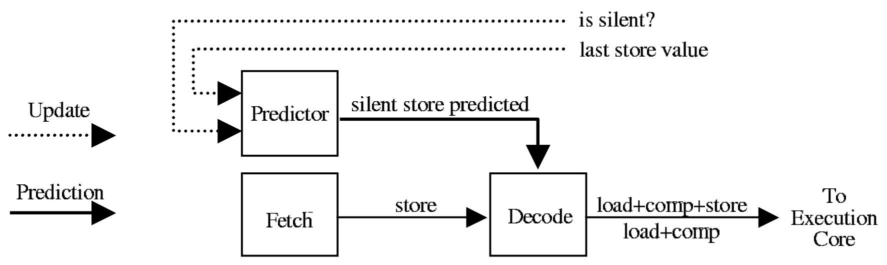


Figure 10. The frontend pipeline with a silent store squashing mechanism. When a store comes into the decoder, it is translated into store / load+compare+store / load+compare depending on a prediction

silent until we actually issue expensive store verify instructions in parallel with the store. In order for a silence predictor to advance from the initial state (no prediction) to higher states (squash stores) without incurring overhead, our predictor mechanism exploits a silent store characteristic that was first report in [4]: stores that consecutively write the same value are more likely to be silent than if they were storing a different value. Figure 7 shows the percentage of a store being silent as a function of whether a static store writes the same value as the last time it is executed. We see that in most benchmarks (except for *li*), over 70% of all silent stores are captured by tracking the last store value. In *vortex, gzip* and *mcf*, over 90% of silent stores write the same value as before. This value locality of static stores can be used to predict silent stores.

Our silence predictor consists of a direct-mapped table where each predictor entry has two n-bit saturating counters and a value history field as shown in Figure 8. The value history field contains only the 8 bits of the register value since initial experiments determined that this was sufficient for providing reasonable prediction accuracy. Storing only 8 value bits significantly reduces the size of the predictor. Whenever the same value is written the confidence counter is increased or conversely decreased, in the same manner as a value prediction confidence table[6]. However, a store may not be silent even though the same value is stored every time by a certain static instruction since a different intervening static store to the same location can alter the stored value. To reduce the number of mispredictions due to aliasing, our predictor mechanism has a dynamic threshold counter that changes according to the prediction outcome history. When the same value is seen for several instances and the confidence reaches the threshold, a store is decoded into a load, compare and unconditional store sequence to verify the store's silence. If it turns out to be silent, the threshold counter is decreased so that the confidence level of a store reaches the threshold easily; If it is not silent, the threshold counter is increased by a fixed penalty and the confidence counter is cleared, which makes it harder for a store instruction to transition out of the "Don't predict" state. Alternatively, when a store is predicted silent, the store is squashed and only a load and a compare instructions are issued until a store verify fails. We show the state diagram of the silence predictor in Figure 9. A store is initially set to "Don't predict" state and the state can advance through "Check" towards "Predict" state based on the store value and silence history.

Figure 10 shows silent store squashing mechanism installed in the pipeline front end. A silence predictor is accessed in parallel with the fetch stage and the decoder generates speculative sequences based on the prediction. Because the prediction is simply made by comparing two fields, it is unlikely that the prediction mechanism is located in the critical path. Since the next prediction can be determined when the predictor is updated, we could alternatively precompute the next prediction at completion time, leaving only a single bit lookup to occur in parallel with instruction fetch. However, we do not feel this is necessary since a 6-bit compare is not expensive. The silence predictor accuracy and performance impact of the silent store squashing will be discussed in Section 4.3 and Section 4.4.

4 .0 Simulation Results

So far, we have discussed the concept and structures for two optimization techniques based on speculative decode. In this section, we quantify the performance benefit from these speculative decode techniques compared with a base machine model. We also show the potential power and energy savings as well as estimated power budget for the speculative decode predictors.

4.1 Machine Model

We used an execution driven simulator of SimpleScalar architecture based on sim-outorder [8]. We enhanced sim-outorder to accurately model finite memory system components such as write buffers, writeback buffers, as well as the scheduling of demand and writeback traffic over the L1 to L2 interface, etc. Because our mechanism focuses on improving memory system performance, modeling these resources accurately is necessary for our results to reflect true performance.

To evaluate the performance benefit from these techniques, we used a realistic, current-generation out-of-order design. The configuration is 4-wide fetch, issue and commit; 32 RUU; 16 LSQ; 4 integer ALUs and 2 integer multipliers. The cache configurations are 64 kB each split I/D L1 and 512 kB unified L2 with latencies of 2, 8 and 50 clocks for the L1, L2 and main memory, respectively. The instruction cache is 2-way associative with a line size of 32 bytes; the data cache is 4-way associative with line size of 16. Store to load forwarding is implemented in the load/store queue with a latency of 2 cycles to match the L1 cache latency. For the branch predictor, we used a state-of-the-art combining branch predictor which is a hybrid mechanism of a GShare and a bimodal predictor: The GShare predictor has 4k entries; the bimodal predictor has 4k entries; and the selector has 4k entries. The branch misprediction penalty is at least 8 clock cycles from the detection of a misprediction to re-execution of the faulty instruction. Therefore, at least 8 cycles are also spent for the recovery from a speculative decode misprediction.

The machine has two fully pipelined load ports and one store port which is mutually exclusive with the load ports. This configuration is based on a multiported cache that is implemented by duplicating a cache array. When a store instruction accesses the cache, no load instruction can access the cache. Conversely, when one or more load instructions access the cache, store instructions cannot be executed. The L1 and L2 caches have a write-back policy. The simulator implements two write buffers outside of the instruction window where committed stores are held until their completion.

4.2 Load/Store Combining predictor Accuracy

In Figure 11, we show the percentage of combined instructions captured by a combining predictor which has 1024 direct-mapped entries. We note that the highest number of the Y-axis is 50%. The percentages of combined instructions range from 2.67% of all memory references in *bzip* to a high of 30.4% in *gcc*, where up to 15% of DL1 cache accesses can be eliminated. Our predictor captures roughly 85% of combinable memory references in all cases except for *go*, where only 64.4% are predicted correctly.

On the other hand, the misprediction rates due to misaligned memory access is hardly visible in the graph because the rates are less than 0.001% of memory references in all benchmarks. The high prediction accuracy provides performance benefit and less power due to lower memory port utilization without any significant misprediction overhead.

It is worthwhile to note that the gcc compiler generates double-word aligned stacks. Double-word aligned stacks are very common in many compilers so that they allow access to doubles in a frame without fear that the operand may be misaligned. This behavior of compilers enable us to easily predict the alignment of combined memory instructions because a lot of load/store alignments are statically determined at compile time.

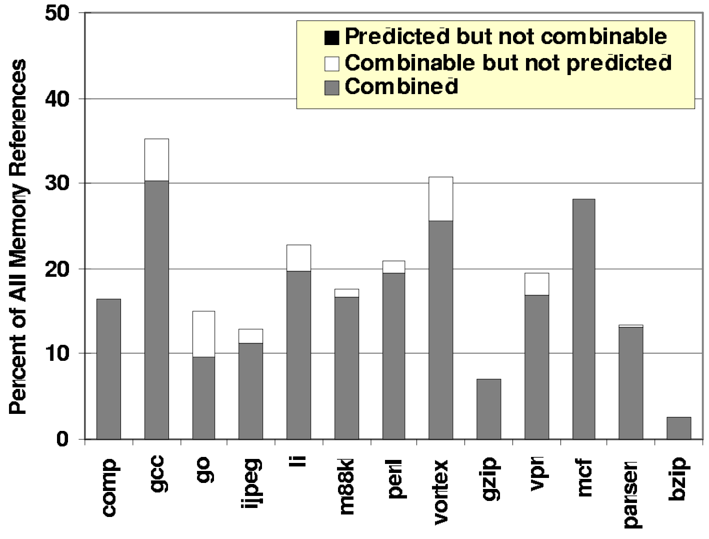


Figure 11. Percentage of combined memory instructions captured by a 1024-entry predictor In most cases, 85% of all combinable memory instructions are predicted correctly. In all benchmarks, the misprediction rates are less than 0.001%.

4.3 Silence predictor Accuracy

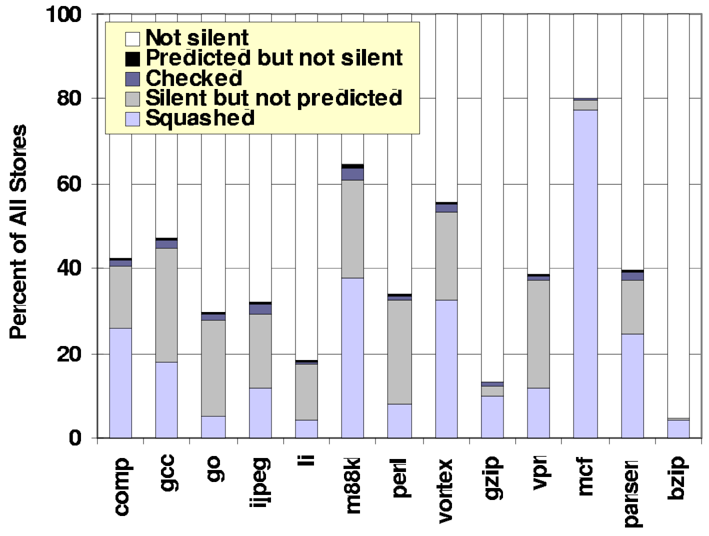


Figure 12. Percent of silent stores captured by a 1024-entry silence predictor.

In Figure 12, we evaluate our silence predictor mechanism for the SpecInt95 and SpecInt2000 benchmarks. The predictor has 1024 direct-mapped entries. 8 bits are used to store the last value written by the store. Confidence and threshold counters have 6 bits each so that they can have maximum values of 63. Initial experiments determined that an effective configuration for threshold counters is initial values of 6 and variance of +4 when silence is mispredicted and -1 when correctly predicted.

Our prediction mechanism correctly predicts a harmonic mean 45.5% of all silent stores, which accounts for 10% of all dynamic stores on average. Because our predictor predicts based on the last store value history, it is interesting to compare the correct prediction count with the same value silent stores shown in Figure 7. In *mcf* and *bzip*, where over 85% of silent stores store the same values, almost all same value silent stores are captured by the predictor. On the other hand, in *go*, only 18.6% of silent stores are predicted correctly even though there are 80% of same value silent stores. One of the reasons is that many non-silent stores in *go* also write the same values as the last time they are executed and these same value non-silent stores decrease the efficacy of our predictor.

The percentage of mispredictions ranges from a low of 0.02% in *bzip* to a high of 0.98% in *m88ksim* as a percentage of all stores. In comparing this result to misprediction rates of combining predictions, we expect that the overhead due to mispredictions will be more significant. However, the potential benefit of silent store squashing is high as well, since squashing silent stores reduces traffic in the memory hierarchy as well as resource contention in the core.

4.4 Performance Improvement of Speculative Decode

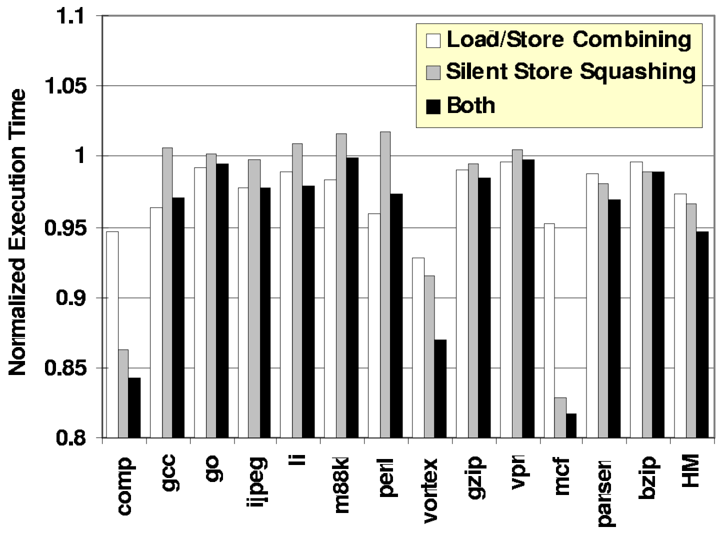


Figure 13. From left to right, the performance improvement of load/store combining, silent store squashing and when both technique are used.

Figure 13 shows normalized execution time with regard to the base machine execution time in load/store combining, silent store squashing and when we use both techniques. Because speculative decode potentially changes the total number of committed instructions and IPC numbers are no longer comparable, the total execution time is an appropriate metric for comparison. The execution time in three configurations is achieved using the predictors described in Section 4.2 and Section 4.3. Note that the lowest number of Y-axis shown is 0.8

In the leftmost bars in Figure 13, load/store combining results show execution time reduction in all cases, ranging from a low 0.44% in *vpr* to high 5.3% in *compress*. A harmonic mean 2.6% of execution time reduction across all benchmarks is achieved. It is important to note that load/store combining does not always improve performance because it delays the use of one of the loaded values until after the rotate instruction. If this use is on the critical path, performance can degrade compared to the base case when both loads can execute in parallel. However, speedups mostly come from the fact that more memory instructions can be issued through the same number of memory ports when the demand for memory ports is high, which eventually improves load latency and reduces the contention in the load/store queue. If load/store combining was applied selectively depending on the memory port contention, or was explicitly avoided when load latency was critical for performance, we would expect greater speedups from combining. We did not consider such optimizations in this initial study.

In silent store squashing, memory intensive programs tend to get more benefit from silent store squashing, as we would expect. However, we see that the distribution of execution time is a quite different across the benchmarks. Performance degrades marginally in half of the benchmarks (a harmonic mean 1.009 of normalized execution time). These slowdowns come from not only the penalty due to mispredictions but also increased resource contention in memory ports and ALUs as well. For example, demand for ALU resources is higher in *m88ksim* than other benchmarks, and is exacerbated by the introduction of store verify operations, so that the slowdown is as bad as time spent for recovery due to silent store mispredictions, even though almost 40% of stores are squashed. In *perl*, the slowdown is even worse than time spent recovering. On the other hand, dramatic performance improvements are observed in *compress, vortex* and *mcf*, among which *compress and vortex* have similar numbers of squashed silent stores. In those benchmarks, the normalized execution times are 0.86, 0.91 and 0.82, which correspond to speedups of 16%, 9.2% and 20.8%, respectively.

Finally, the rightmost bars in Figure 13 show the performance when both load/store combining and silent store squashing are applied simultaneously. In this configuration, when two stores are predicted as combinable and silent, they are speculatively decoded into a merge, a double-word load, and a compare. If only one of the combinable stores is predicted as silent, silent store prediction is ignored so that we can get fewer mispredictions because combining prediction has a lower misprediction rate. We obtain speedups even in the benchmarks which show slowdowns in silent store squashing because speedups achieved due to load/store combining exceed silent store squashing. It is interesting to note that in *li*, the combined benefit is greater than when only load/store combining is applied even though silent store squashing results in slowdown. This implies the benchmark benefits from the combined silent store verify and the overhead is reduced by applying both techniques. In *compress, vortex* and *mcf*, where silent store squashing provides most speedup, significant speedups also occur when both techniques are applied. We achieve execution time reductions ranging from 0%(*m88ksim* and *vpr)* up to 18.3% in *mcf*, which corresponds to 22.4% speedup. On average, 5.3% execution time reduction is achieved by these two speculative decode techniques.

4.5 The impact of speculative decode mispredictions on performance

Because recovery from mispredictions in speculative decode relies on the branch recovery mechanism that drains the pipeline and refetches instructions, we expect that speculative decode performance would correlate with misprediction count. Table 2 shows the effect of speculative decode mispredictions compared with branch misprediction count. The second column shows the branch prediction rate of our base machine. Unsurprisingly, the combining branch predictor that we use has very high prediction rates.

The third column shows the fraction of total execution time that is spent due to branch mispredictions on the base machine. In the fourth column, we show the estimated time that will be spent due to speculative decode mispredictions if speculative decode is applied to the base machine. In this estimation, we took the load instruction latency needed for store verify into considerations. As we can see, the effect of speculative mispredictions are found to be unimportant in most benchmarks because the total misprediction count of speculative decode is lower than branch mispredictions. In most benchmarks, the extra recovery time is less than 10% of time needed due to branch recovery. In *vortex* which shows very accurate branch prediction rates, although time spent on recovery significantly increases by 82%, the performance boost from speculative decode is so great that it still shows speedups.

Even though the trend is that pipelines are getting deeper in order to attain the higher clock speeds, the effect of mispredictions is not likely to be a critical drawback on the performance of speculative decode because the misprediction count is much lower than branch mispredictions. Alternatively speculative decode may be able to use a selective invalidation mechanism such as Replay that is implemented in the Intel Pentium-IV[13]. This is possible because the control flow does not change during speculative decode misprediction recovery.

Table 2: Comparison of branch and speculative decode mispredictions

Benchmarks	Branch prediction correct rate	The fraction of execution time spent on branch recovery on a base machine	The fraction of extra time to be spent on speculative decode recovery
compress	93.0%	14.33%	1.41%
gcc	91.7%	18.06%	1.30%
go	83.3%	27.23%	0.52%
ijpeg	95.5%	13.27%	0.83%
li	94.5%	15.43%	0.67%
m88ksim	96.7%	12.62%	1.86%
perl	96.3%	9.30%	1.46%
vortex	98.3%	2.39%	1.96%
gzip	94.6%	13.12%	0.13%
vpr	89.1%	21.77%	1.20%
mcf	90.4%	7.86%	0.04%
parser	93.9%	14.45%	0.62%
bzip	98.0%	3.84%	0.08%

4.6 Power/Energy Reduction due to Speculative Decode

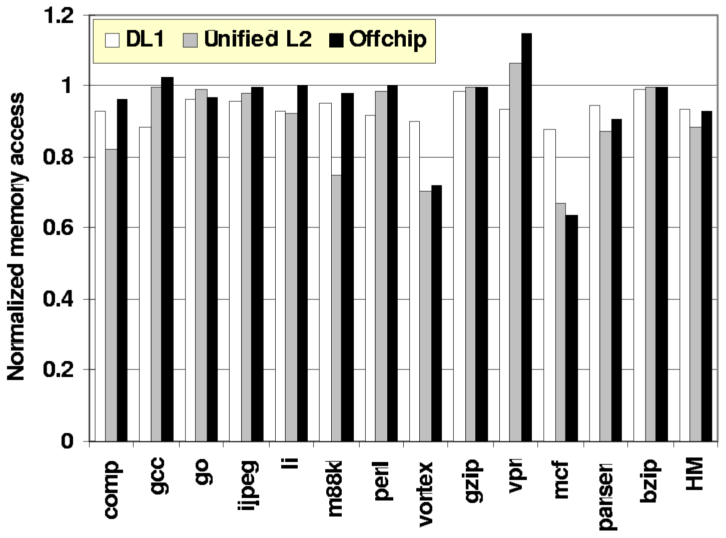


Figure 14. From left to right, DL1, L2 and offchip accesses. In most benchmarks, memory access counts are reduced. In *vpr* and *gcc*, L2 and offchip accesses increases due to the secondary effect on the replacement policy.

In speculative decode the logical semantics of the program do not change and the program behaves in the same way, but the observed execution of the program is changed toward reducing memory accesses, which can result in lower power consumption. Figure 15 shows the normalized access counts of, from left to right, a DL1 cache, a uni-

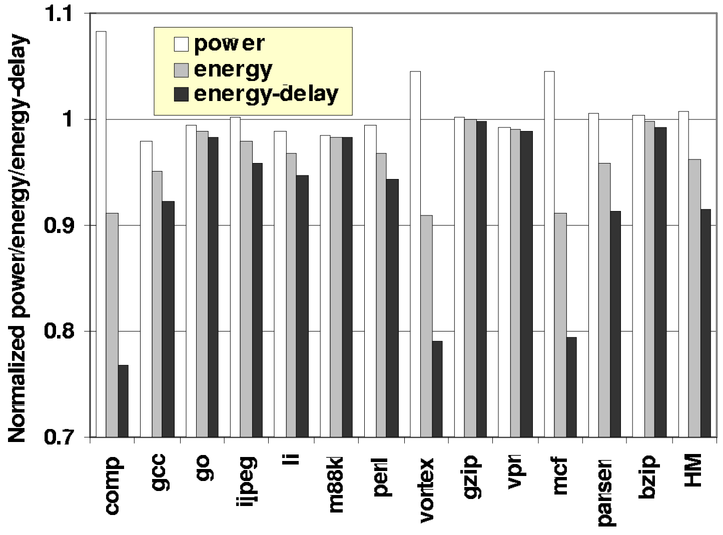


Figure 15. Power, energy and energy-delay estimations when clock gating and 10% leakage power are assumed. Note that power consumption due to speculative decode logic and predictors is not included.

fied L2 and offchip due to speculative decode when load/store combining and silent store squashing are performed simultaneously. Load/store combining can reduce the DL1 cache accesses which account for a significant portion of power consumption: Lower level cache and offchip accesses are reduced by silent store squashing due to writeback reductions. Most benchmark shows measurable reduction in memory accesses at each memory level. It is interesting to note that lower level cache accesses are increased in *gcc* and *vpr*, where cache lines that should have been marked most recently used are frequently replaced because higher level memory does not touch those entries due to writeback elimination. Because the number of lower level memory accesses are relatively lower than DL1 accesses, more L2 accesses do not necessarily mean higher power consumption.

In order to determine how effectively speculative decode reduces power and energy consumption, we used Wattch[9], an execution driven, architectural level power and energy consumption simulator. In Wattch, peak power consumption of each unit in a processor are estimated based on the size, capacitance, technology and the structure type and then the total power and energy consumption is calculated by the access counts to all units in a processor. Unfortunately, Wattch does not accurately model the power consumptions in lower level caches because the original simplescalar does not exactly model lower level traffic. We modeled the power consumption in lower level memory by implementing a more exact memory system model. However, our extended model, like Wattch, does not estimate the power consumption due to offchip accesses.

Figure 15 shows relative power, energy consumption and energy delay when load/store combining and silent store squashing are applied compared with a base machine across all benchmarks. This data was collected using the cc3 option of Wattch in which conditional clocking is assumed so if only portion of a unit's ports are accessed, the power is scaled; if a unit is not used, the unit dissipates 10% of its maximum power as leakage power. We note that additional power consumption due to speculative decode and predictors is not included in the graph for the following reasons: first, we cannot estimate how much more power the decode logic will consume due to speculative decode. Second, we cannot estimate the power consumption due to predictor tables. Since the estimated size of the predictor structure that we use is as low as 3.9Kbytes for a load/store combining predictor and 2.4Kbytes for a silence predictor, and a real implementation of these predictors could be much more power-efficient[11][12], we expect the additional power consumption will not be drastic. However, careful analysis which is beyond the scope of this paper is required before a definitive conclusion can be reached.

We show the power consumption per clock cycle in the leftmost bars. The average power consumption is slightly increased by a harmonic mean of 0.8%. *compress*, *vortex* and *mcf* show increases up to 8.3% because power is a function of work and time and the execution time of those benchmarks significantly decreases. On the other hand, in *gcc, go, li, m88ksim* and *vpr*, the power consumption is reduced even though their execution time is decreased or almost not changed. This is because memory instructions are speculatively decoded into other instructions that consume less power. In the case of energy consumption, shown in the middle bars, the average energy reduction is 4.4% across all benchmarks, ranging from 0% to 8.9% in *vortex*. It is interesting that we see energy benefit even in the benchmarks that show more power consumption because the performance improvement exceeds the increased power consumption. Finally, the rightmost bars show the energy-delay product improvement with a harmonic mean 9.8%.

This result suggests another possibility of speculative decode for dynamic power and speed adaptation. If we have a mechanism that dynamically changes its optimization level depending on the workload and power budget, the processor can be optimized toward performance improvement or more power savings without significant core changes. For example, if the temperature sensor detects a processor is too hot, instructions can be decoded into a slow but low power-consuming stream; when performance is critical, speculative decode mechanism can increase the optimization level toward performance at the expense of increased power. We plan to investigate such adaptive mechanisms in future work.

5 .0 Conclusions and Future Work

We make five major contributions in this work. First of all, we introduce the concept of speculative decode as a microarchitectural technique for enabling unsafe code transformations that enhance performance and/or reduce energy consumption without significant changes to an existing processor core. Second, we demonstrate that load/store reference combining implemented within the speculative decode framework can provide up to 5.6% (average 2.7%) performance improvement. This is enabled by our third contribution: a novel compact predictor design that accurately identifies combinable instructions with coverage of 89.6% and a misprediction rate of only 0.001%. Fourth, we show that silent store detection and verification can be implemented within speculative decode with minimal overhead, resulting in performance improvements up to 20.8% (average 3.4%). This is enabled by our fifth and final contribution: a novel value-history-based predictor design that correctly identifies 45.5% of silent stores and mispredicts the silence of only 0.41% of stores. These two techniques result in performance improvements of 5.6% and energy savings of 4.4% on average.

Our study focuses primarily on performance benefits that are attainable through speculative decode. We also evaluate power consumption and find that both the energy and energy-delay metrics improve as secondary benefits of our approach. However, we do not exhaustively evaluate the opportunities available in the spectrum of trade-offs between minimizing power consumption and maximizing performance. It is worth noting that the presence of mechanisms like the ones proposed in this paper creates additional opportunities for processor designers seeking to target the same processor core for multiple market segments, some of which may require high performance (e.g the desktop/gaming market) while others may require low power consumption (e.g. the portable or embedded market). In fact, this market segment differentiation can be trivially accomplished by adjusting the parameters that guide the speculative decode transformations. We plan to study this opportunity in future work by demonstrating the spectrum of power/performance design points that are attainable with such minor changes.

Finally, we argue that much more significant benefits could be derived from the speculative decode approach if the implementation instruction set (I-ISA) provides a richer set of semantics for manipulating the control-path constructs that exist within a modern out-of-order processor core. To the best of our knowledge, current real I-ISAs closely resemble canonical RISC load/store instruction sets. In our view, this class of instruction sets are data-path-centric. That is, the original motivation for their definition was to simplify the interface between the programmer or compiler and the available data path hardware (i.e. ALUs and data cache) by exposing that data path explicitly, rather than obscuring control over it behind instructions with complex semantics. Unfortunately, the predominant source of complexity in modern processor design is no longer the data path, as it was twenty-some years ago when RISC instruction sets were first conceived. Rather, most of the complexity is buried in complex control path components that are implicitly manipulated according to data-path instruction semantics. We advocate exposing these complex control paths to the instruction set in a manner analogous the exposure of data path components that occurred when RISC instruction sets (and, by extension, VLIW instruction sets) were introduced. This effort to unmask control-path components and structures to the instruction set is a subject of current and future work.

References

[1] H. W. Cain, K. M. Lepak and M. H. Lipasti, *A Dynamic Binary Translation Approach to Architectural Simulation,* in *Workshop on Binary Translation* in *PACT*, October 2000.

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

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

[4] 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.

[5] K. M. Lepak and M. H. Lipasti. *Silent Stores for Free*, to be appearing in Proc. of 33th *MICRO*, December 2000.

[6] M. H. Lipasti and J. P. Shen. *Exceeding the Dataflow Limit via Value Prediction*, in Proc. of 29th *Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

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

[8] 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.

[9] D. Brooks, V. Tiwari and M. Martonosi, *Wattch: a Framework for Architectural-Level Power Analysis and Optimizations*, in Proc. of 27th *International Symposium on Computer Architecture*, June 2000.

[10] C. May, E. Silha, R. Simpson, and H. Warren, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, 2nd ed. Morgan Kaufmann, San Francisco, California. 1994.

[11] C. Su and A. M. Despain, *Cache Design Trade-offs for Power and Performance Optimization: a case study*, in Proc. of *International Symposium on Low Power Design*, April 1995.

[12] A. Malik, B. Moyer and D. Cermak, *A Low Power Unified Cache Architecture Providing Power and Performance Flexibility*, in Proc. of *International Symposium on Lower Power Electronics and Design*, 2000.

[13] Intel Corp. *Intel Pentium 4 Processor Optimization Reference Manual*, http://developer.intel.com/design/pentium4/manuals/24896601.pdf. November 2000.

[14] Intel Corp. *Intel Architecture Optimization Reference Manual*, http://developer.intel.com/design/PentiumII/manuals/24512701.pdf, September 1999.

[15] Advanced Micro Devices Inc., *AMD-K6 Processor Data Sheet*, http://www.amd.com/K6/k6docs/pdf/20695.pdf, March 1998.

[16] C. Moore, *POWER4 System Microarchitecture*, Proc. of *Microprocessor Forum*, Cahners MicroDesign Resources, 2000.

[17] M. H. Lipasti, C. B. Wilkerson and J. P. Shen, *Value Lacality and Load Value Prediction*, in Proc. of 7th *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[18] G. Kane and J. Heinrich, *MIPS RISC Architecture*, 2nd ed. Prentice Hall, Englewood Cliffs, NJ, 1991.

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

[20] IBM Microelectronics and Motorola Inc., *PowerPC 604 RISC Microprocessor User's Manual*, 1994.

[21] T, Shanley, *Pentium Pro and Pentium II System Architecture*, 2nd ed., MindShare Inc., Addision Wesley Publishing Company, April 1997.

[22] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli and J. Yates, *FX!32 - A Profile-directed Binary Translator*, *IEEE MICRO*, 18(2), March-April 1998.

[23] K. Ebcioglu and E. Altman, *DAISY: Dynamic Compilation for 100% Architectural Compatibility*, in Proc. of 24th *Interna-*

tional Symposium on Computer Architecture, June 1997.

[24] Transmeta Corp., *The Technology behind the Crusoe Processor*, Tech. Report, January 2000.

[25] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, *Dynamic and Transparent Binary Translation*, *IEEE Computer*, Vol. 33, No. 3, March 2000.

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

[27] Intel Corp., *P6 Family of Processors - Hardware Developer's Manual*, 1998.

[28] E. Rotenberg, S. Bennett and J. E. Smith, *Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching*, in Proc. of 29th *international symposium on Microarchitecture*, 1996.

[29] C.J. Newburn, *Personal communication regarding Intel Pentium-IV development*, September 2000.