Characterization of Silent Stores

Gordon B. Bell
University of Wisconsin--Madison
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 18, 2001

1 .0 Introduction

Value locality, a recently discovered program attribute, has been the subject of much recent research [7, 8, 9, 11]. Through the exploitation of value locality, a limit which was once considered unsurpassable in modern computing (the so-called “dataflow limit”) can now be exceeded. Until very recently, most work has proposed using value locality to accelerate the input portion of a computing pipeline, or the “means of computing” through value prediction. However, some very recent work tries to determine if value locality can also accelerate the output portion of a computing pipeline, or the “end of computing” through exploiting store value locality [7].

Since the idea of store value locality is relatively new, a brief review of the idea is in order. Store value locality is the idea that a store is writing the same data value to the memory system that was previously located there or is trivially predictable. Therefore, since the state of memory is not changed by the store (or is changed only trivially), it is unnecessary. Intuitively, if we can remove the store at some point in the program execution (either statically at compile time, or dynamically at run time) some potential benefit can be gained in execution time and/or code size.

This work focuses on characterizing silent stores (referred to throughout this work as store value locality, or SVL). A silent store is defined as a memory store operation that does not change the contents at that location. Because they do not change the state of memory, all silent stores can be eliminated without affecting program correctness.[1] A store verify is used to determine if a store is silent. It consists of up to three operations--a load, compare, and possible store. Before a store commits, the value at that store’s address is loaded from memory. If this value is identical to the value the instruction will store, the store is squashed, or removed from program execution. Otherwise the store continues normally and writes the new value to memory. Silent stores can be squashed at the level they are identified--either dynamically as they execute, or statically at the level of code generation. It is shown in [7] that silent stores constitute 18%-64% of total program stores in a variety of benchmarks and can consume significant processor resources, therefore it is highly desirable to removal them.

2 .0 Motivation for Exploiting SVL

Reducing Cache Writebacks: Most modern microprocessors use writeback caching to improve performance [6]. On a cache miss, if the line being replaced is dirty, it is generally copied into a writeback buffer (WB), the demanded line is loaded, and then the dirty line is written back whenever there is sufficient cache bandwidth. This means that many misses result in two memory accesses--one to load the demanded line, the other to write back the dirty line. If we can reduce the number of writebacks necessary by having fewer dirty lines, we will require less bandwidth to the next layer of memory hierarchy. Although more extensive multiprocessor considerations for removing silent stores are given in [7], as a brief synopsis, we can imagine that reducing the number of writebacks can also cause fewer cacheline invalidates in an invalidation-based coherence protocol or fewer updates in an update-based protocol. It has also been shown that exploiting certain properties of silent stores (or stochastically silent stores [7]) can also reduce the amount of true and false sharing that occurs in multiprocessor systems.

Reducing the Number of Writeback Buffers: As described in the previous section, in order to allow the demanded miss to be serviced before writing the replaced line back, we need to buffer the dirty line in a writeback buffer. If we can reduce the number of writebacks, we can also potentially reduce the number of writeback buffers used in a processor design to maintain acceptable performance and to save chip area and design complexity.

Reduction in CAM Complexity: Due to memory consistency model constraints which dictate the order in which memory operations take place [1], complex Content Addressable Memory (CAM) structures are required for fully associative look-ups in order to maintain program correctness. For example, consider a strongly consistent architecture, where all memory operations must appear to occur in logical program order. Let us examine what needs to happen in an architecture with store to load forwarding from the load-store queue (LSQ). If a load occurs to a location that has been recently stored to by the program, the processor must either forward the store value in the store queue or wait for the store to commit to memory. Suppose further that there are multiple store operations to the target load’s address in the LSQ. The processor must be able to forward from the most recent

1. There are some cases where multiprocessor semantics can complicate this process.

matching entry in the LSQ (i.e. the LSQ is a priority associative queue). We can imagine that these structures can require many associative address lookups, and we know that building a fast CAM is difficult due to circuit constraints. If we can reduce the number of stores, we can potentially reduce the size of the store queue and hence the required number of associative lookups..

Performing Write Operations Is Comparatively Difficult: As modern processors become wider issue, memory bandwidth is becoming a problem, not only in terms of cache latencies, but also in terms of having a sufficient number of cache ports to perform the memory operations. It has been demonstrated [6] that obtaining multiple read ports into a memory structure can be obtained through many schemes, the simplest of which is duplication of the memory resource. However, performing multiple writes is generally more complicated, requiring interleaving or multiported memory cells. Given this, it should be obvious that removing writes (or trading a write for a read with a store verify) can increase performance or allow a reduction in complexity of the memory system and maintain equivalent performance.

3 .0 Related Work

Lepak and Lipasti [7], as well as Molina et. al. [11] introduced the notion of silent stores and showed that they constitute a non-trivial percentage of overall stores. Given empirical evidence that silent stores do exist, this paper explores why they exist and discusses several of their primary characteristics. Although specific optimizations to exploit silent stores are not directly proposed here, we expect that first understanding why they occur will naturally lead to novel optimizations and future research.

Calder et al. [4] discussed the concept of a silent store in terms of hoisting loads and dependent instructions above loop bodies. They proposed a hardware mechanism that extended the Memory Disambiguation Buffer (MDB) by not only checking if an instruction stores to the address of the hoisted load, but also if the value has been changed (i.e. the store is silent).

The notion of store removal is not new in compiler literature (e.g. [2]). A common dataflow optimization is to remove dead stores, or stores followed by a second store to the same address without an intervening load from that address. Because the stored value is overwritten before it is used, all dead stores can be eliminated. Although superficially they may seem similar, there are several important distinctions between dead stores and silent stores. The most obvious is that a store does not have to be silent in order for it to be dead. Being dead simply implies that its value is overwritten before it is used; it makes no difference if the new value is the same as the old. A second difference involves which store is eliminated. If two stores write the same value to the same address, the second store is silent and it is removed. On the other hand, if by overwriting a location, a store causes another store to be dead, the first store is removed. This means that silent stores can be identified much later than dead stores. Because knowledge of future instructions is required, dead store identification is usually limited to compilers, where silent stores can be identified and eliminated dynamically in hardware. This offers significant flexibility--a store can be squashed if it is silent even once, while requirements for removing dead stores can be considered more strict.

4 .0 Source Code Analysis

4.1 Code Size/Efficiency

If we can remove unnecessary instructions from the program entirely or can take them off the critical path of execution, we can either gain some efficiency in code size, performance, or both. In many cases, not only can silent stores be removed from program execution, but also related instructions that are used in the store’s data or address computation. Consider a FORTRAN code segment from the SPECFP-95 benchmark *mgrid* in Figure 1. In our compiler, this statement compiles to 107 machine instructions and is silent over 34,000 times. Thus if we could eliminate the computation required when just this single static store is silent, over 3.6 million instructions could be removed from program execution, or about 2.4% of the total number of dynamic instructions. Of course, this assumes that all side-effects and exceptions generated by the removed code are honored in order to maintain consistent program behavior.

4.2 Stores Likely to be Silent

Given the frequency and impact of silent stores, the first question to ask is ‘Why do silent stores exist at all?’.

```
R(I1,I2,I3) = V(I1,I2,I3) - A(0)*( U(I1,I2,I3))
- A(1)*(U(I1-1,I2,I3) + U(I1+1,I2,I3) + U(I1,I2-1,I3)
+ U(I1,I2+1,I3) + U(I1,I2,I3-1) + U(I1,I2,I3+1))
- A(2)*(U(I1-1,I2-1,I3) + U(I1+1,I2-1,I3)
+ U(I1-1,I2+1,I3) + U(I1+1,I2+1,I3) + U(I1,I2-1,I3-1)
+ U(I1,I2+1,I3-1) + U(I1,I2-1,I3+1) + U(I1,I2+1,I3+1)
+ U(I1-1,I2,I3-1) + U(I1-1,I2,I3+1) + U(I1+1,I2,I3-1)
+ U(I1+1,I2,I3+1)) - A(3)*(U(I1-1,I2-1,I3-1)
+ U(I1+1,I2-1,I3-1) + U(I1-1,I2+1,I3-1)
+ U(I1+1,I2+1,I3-1) + U(I1-1,I2-1,I3+1)
```

FIGURE 1. *mgrid:* Eliminating this expression removes not only a store, but also over 100 other static instructions.

What causes a compiler to insert stores that have no effect on the state of the processor? One explanation lies in the nature of program generality. Programs are often designed to operate on a variety of data input sets, and while stores may update memory contents when executed with some inputs, they may not with others [15]. Consider an example from the *m88ksim* benchmark (Figure 2).

```
for (i = 0; i < 32; i++){
 m88000.time_left[i]-=MIN(m88000.time_left[i], time_to_kill);
}
```

FIGURE 2. *m88ksim*: Frequently the store in all 32 loop iterations is silent, allowing the entire loop to be bypassed.

This instruction continually decrements an array element by the minimum of itself and the unsigned value *time_to_kill*. Clearly, at some point the value stored in memory will converge to zero, and every subsequent store to that location will continue to store zero. In the input data sets included with SPECINT-95, this convergence occurs quickly and in over 95% of all executions this store is silent. Because this particular piece of code is heavily executed, this single static store is responsible for over half of the benchmark's 5 million dynamic silent stores.

Furthermore, not only is this store frequently silent, but 27% of the time the stores in all of the 32 loop iterations are silent. Thus, if it is known that every element in the array is zero, the entire loop can be skipped. This could be exploited in a technique similar to memoization [5, 12] or dynamic instruction reuse [15], in which the values of the array are used to index a table whose outcome conditionally determines if the loop should be executed. This leads to a substantial savings, as not only would the silent stores be skipped, but also the call to the *MIN* function, the potential load of *time_to_kill*, the subtraction of *time_to_kill* from *m88000.time_left[i]*, the loop induction variable calculations and the backwards branch to the start of the loop. By just skipping the loop when all array elements are zero, 9.6 million fewer instructions need to be executed, or about 11% of the total dynamic instruction count.

This example leads to four classifications of dynamic silent stores based on their previous execution (similar to general store classifications in [7]):

Same Location, Same Value: A static silent store stores the same value to the same location as the last time it was executed. This is the case in the *eval* function from the *perl* benchmark in Figure 3.

Each iteration through this loop processes an argument and stores the argument's flags in the stack-allocated temporary *argflags*. Many arguments have *arg_flags* equal to zero, thus the store is silent 71% of the time.

```
    for (anum = 1; anum <= maxarg; anum++) {
     argflags = arg[anum].arg_flags;
```

FIGURE 3. *perl*: Same Location, Same Value.

Different Location, Same Value: A silent store stores the same value to a different location as the last time it was executed. This situation often occurs when an instruction is storing to an array indexed by a loop induction variable, such as the *m88ksim* example just presented and the Figure 4 example from *go*. The bottom three stores initialize every location in the arrays that represent the game board to the same value (zero, or equivalently, FALSE). However the stores' addresses are derived from the loop index, and thus are different in each iteration. Often the values on the board already contain zero (or FALSE), so these three stores are silent 86%, 43%, and 77% of the time, respectively.

```
for(x = xmin; x <= xmax; ++x)
            for(y = ymin; y <= ymax; ++y){
                   s = y*boardsize+x;
                   ...
                   ltrscr -= ltr2[s];

                   ltr2[s] = 0;
                   ltr1[s] = 0;
                   ltrgd[s] = FALSE;
            }
```

FIGURE 4. *go:* Different Location, Same Value (last 3); Same Location Different Value (*ltrscr*).

Same Location, Different Value: A silent store stores a different value to the same location as the last time it was executed. The store to *ltrscr* in Figure 4 is of this type. *ltrscr* is a global variable (thus its location does not change), but stores a different element (and thus a potentially different value) of the *ltr2* array. If another instruction updates *ltr2* between successive executions of this function, the store has the potential to be silent in the first loop iteration. Although this store is silent 86% of the time, 98% of the time that it is silent consecutive elements of *ltr2* are zero and the store is usually Same Location, Same Value.

Although silent stores of this category are generally rare, they usually exist due to one of several circumstances. Many times the silent value was previously stored to memory by another instruction, corresponding to message-passing store value locality introduced by Lepak and Lipasti in [7]. More often, Same Location, Different Value silent stores are caused by compiler and architecture conventions that dictate how stack frames are used. For example, when a function is invoked in a callee-save register convention, it immediately saves the contents of a predefined set of registers onto its stack frame (although it does not necessarily have to if it will not modify those register contents before retuning). If these register contents do not change between calls from within the same function, each called function will silently store them onto the same stack frame. Similarly, a function often stores its caller's frame pointer and its return address onto its stack frame. Because these values do not change between successive function calls from the same caller, they too are likely to be silent. A final case of silent stores of this category involve architectures that use the stack to pass function arguments (either by passing arguments explicitly on the stack or by saving register contents to memory before using them to pass arguments).

Consider the fabricated code example in Figure 5. The top part shows main with three calls to two functions: foo_1 and foo_2. The bottom part shows assembly code for each function. When foo_1 is called, an activation record is allocated on the stack and the frame pointer, return address, and callee-saved registers are stored to (and later loaded from) the stack frame. Because foo_2 is called from the same function (main), the activation record it allocates overlays directly on top of the one de-allocated by foo_1. Thus boldfaced stores are silent. However, these static stores within foo_2 are same location different value (except for *$fp* which is same location same value) because foo_2 was previously invoked from a different location at the same call depth, as shown in the initial call to foo_2(d,e,f).

Different Location, Different Value: A silent store stores a different value to a different location as the last time it was executed. Examples include nested loops that store a set of values into an array in the inner loop. The *xlsave* procedure that saves a set of nodes onto the stack in the *li* benchmark is one such case (Figure 6). In the first execution of *xlsave*, each iteration of the loop sets *nptr* equal to the next function argument (different value)

```
void main(){
 int a=1; int b=2; int c=3;
 int d=4; int e=5; int f=6;
 foo_2(d,e,f);
 foo_1(a,b,c);
 foo_2(a,b,c);
}
**** void foo_1(int x, int y, int z) ****
... allocate new stack frame ...
sw $fp,32($sp) // store old frame ptr to stack
sw $16,16($sp) // R16 is a callee-save register
sw $17,28($sp) // R17 is a callee-save register
... foo_1 function body ...
lw $fp,32($sp) // restore frame ptr
lw $31,36($sp) // load return address
lw $16,16($sp) // restore callee-save register
lw $17(28($sp) // restore callee-save register
... restore previous stack frame ...
jal $31 // return

**** void foo_2(int x, int y, int z) ****
... allocate new stack frame ...
sw $fp,32($sp) // store old frame ptr to stack
sw $16,16($sp) // R16 is a callee-save register
sw $17,28($sp) // R17 is a callee-save register
... foo_2 function body ...
lw $fp,32($sp) // restore frame ptr
lw $31,36($sp) // load return address
lw $16,16($sp) // restore callee-save register
lw $17(28($sp) // restore callee-save register
... restore previous stack frame ...
jal $31 // return
```

FIGURE 5. Same Location, Different Value.

```
NODE ***xlsave(NODE **nptr,...){
   ...
   for (; nptr != (NODE **) NULL; nptr = va_arg(pvar, NODE **)){
                ...
                *--xlstack = nptr;
                ...
```

FIGURE 6. *li:* Different Location, Different Value.

and decrements the store address (different location). However, if subsequent calls to *xlsave* store the same set of nodes to the same starting stack address, each store instruction will be silent (as it is 48% of the time).

A store's probability of being silent based on these categories is shown in Figure 7. In all of the benchmarks, instructions that consecutively store the same value are more likely to be silent than if they were storing a different value. An important corollary to this graph is Figure 8, which adds instruction frequency information by showing each silent store category's contribution to total silent stores. Because stores that have a high likelihood of being silent are also executed frequently (this will be shown in Section 5.2), they represent a significant portion of all dynamic silent stores. One could imagine using such categories as an aid to predict if a store will be silent or not early in the pipeline (and thus if a store verify should be performed). Such a mechanism is not discussed here and remains an opportunity for future work.

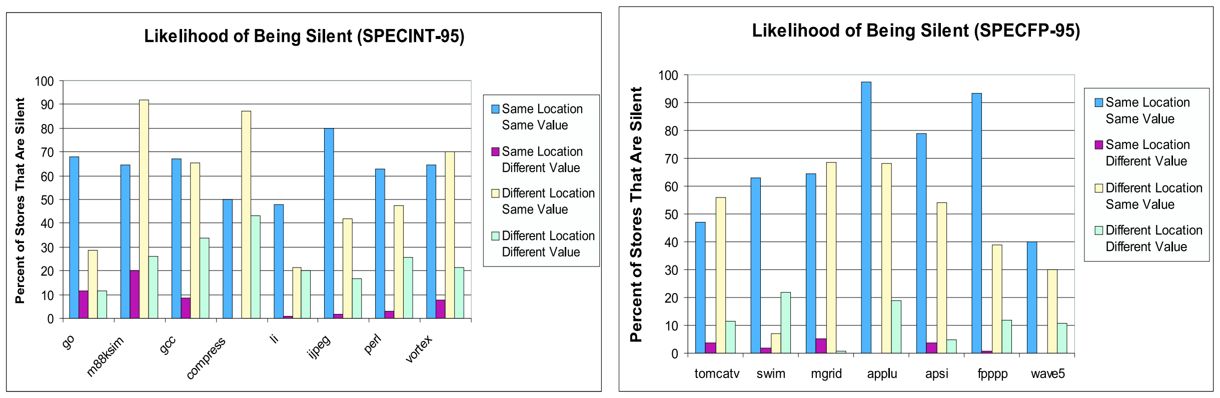


FIGURE 7. Probability of a store being silent as a function of store category.

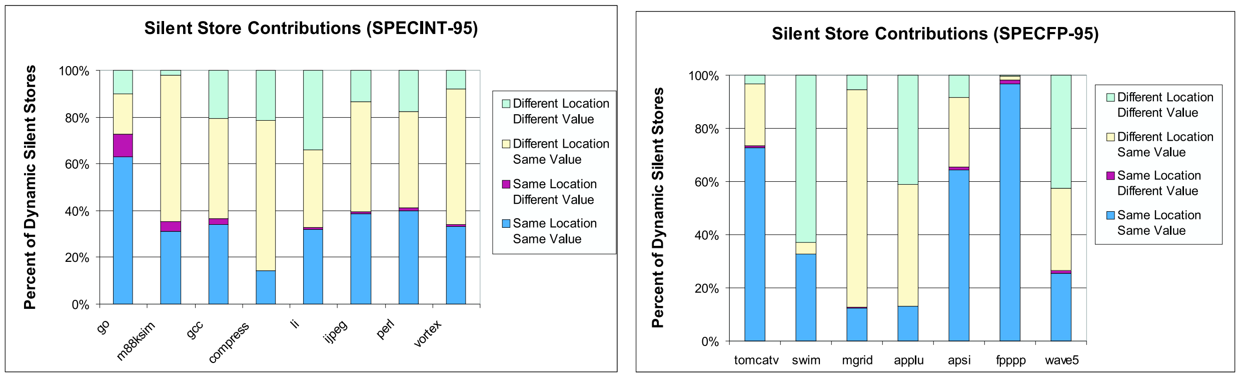


FIGURE 8. Total silent store contributions as a function of store category.

4.3 Critical Silent Stores

Although a silent store does not update the contents of stored data, it does set a line's dirty bit in a write back cache configuration. If the dirty bit would not have been set otherwise, this writeback could be avoided by squashing the silent store.

Definition. A *critical silent store* is a specific dynamic silent store that, if not squashed, will cause a cacheline to be marked as dirty and hence require a writeback.

This definition applies for each lifetime of the given cacheline in the cache (the time between each allocation and replacement). Each cacheline lifetime may have zero to *n* critical silent stores. Trivially, if there are no stores to the line, there are no critical silent stores either. Similarly, if there is even one non-silent store, there are no critical silent stores. However, if there are one or more silent stores to the line and no non-silent stores, the former set of silent stores is defined as critical, since failing to squash any of them will result in a writeback. Put more simply, it is sufficient to only squash the critical silent stores to obtain maximal writeback reduction. Squashing non-critical silent stores has no benefit (in terms of writeback reduction) because the line will be written back when it leaves the cache anyway because a non-silent store wrote the line during its lifetime. Furthermore, squashing non-critical silent store misses can actually degrade performance because we incur the load and compare overhead of a store verify without any compensating reduction in writebacks. This problem is worse in multiprocessors. A non-critical silent store is replaced with a store verify (read), but a subsequent non-silent store to that cache line will require that the line be upgraded from a shared to modified state (requiring an upgrade/invalidate bus transaction). We can be sure this upgrade will in fact take place because of the definition of a critical silent store. If the non-critical silent store suffered a cache miss and was not squashed, the line would have been brought into the cache with a read-with-intent-to-modify transaction (hence obtaining the line directly in modified state) and the upgrade message would not be required. Thus squashing a non-critical silent store miss leads to an additional address bus transaction--namely the line upgrade when a non-silent store to the line occurs.

We determine the critical silent stores in the simulator in the following way: When a line enters the cache, we allocate a list (called the candidate list) to hold all stores to that line during its lifetime. When a store accesses the line, we add it to the candidate list for that line. When the line leaves the cache, we check if the line is dirty. If it is dirty, we know all stores to the line are non-critical (because a writeback of the line occurs anyway). If it is not dirty, if any stores exist in the candidate list, we know that all of these stores must have been silent, and also since a writeback is not occurring, they are all critical (by definition). We then account for the stores properly in the global critical silent store tracking structures, as the current lifetime for the line has ended. Finally, we clear the candidate list.

Table 2 presents the number of writebacks in each SPEC benchmark without silent store squashing (baseline case), the percent of those writebacks that are eliminated when silent stores in all levels of the memory hierarchy are squashed, and the percentage of silent stores that are critical for three different cache configurations (32KB cache with 32B lines, 8KB cache with 32B lines, 8KB cache with 8B lines). We see that, in some cases, selectively squashing only a fraction of all silent stores can dramatically reduce the total number of writebacks incurred by a program (for example in vortex 79% of the total writebacks can be eliminated by squashing only 39% of all silent stores). We also see that decreasing the cache size increases the number of silent stores that are critical because lines spend less time in the cache before being replaced. This highlights the fact that our definition of critical stores depends on cacheline lifetime, and can be influenced by such factors as cache size, associativity, and replacement policy. Since lines are replaced more frequently, there is not as much opportunity for non-silent stores to write to them and the lines have a better chance of leaving the cache unmodified. Decreasing the line size also increases the number of critical silent stores for the same reason that decreasing line size helps to eliminate false sharing. If there are few words per cache line, there is less chance that one of them will be written to by a non-critical silent store. However, it is important to remember that, even though decreasing line size and total cache size may yield a greater percentage of writebacks that can be eliminated by squashing, the total number of writebacks increases as well

.

Table 1: Writeback reduction due to critical silent stores in various cache configurations.

	32 KB / 32 B			8 KB / 32 B			8 KB / 8 B		
SPEC95 Benchmark	Baseline WB / instr (x 10^{-3})	% WB reduced by squashing	% of silent stores that are critical	Baseline WB / instr (x 10^{-3})	% WB reduced by squashing	% of silent stores that are critical	Baseline WB/instr (x 10^{-3})	% WB reduced by squashing	% of silent stores that are critical
go	1.04	9.2%	4.5%	5.57	6.4%	6.6%	6.51	14.0%	10.3%
m88ksim	0.30	56.0%	14.0%	0.36	59.0%	15.0%	1.05	64.2%	15.4%
gcc	1.14	18.8%	13.9%	2.67	23.3%	8.9%	5.58	30.1%	17.6%
compress	11.5	58.7%	71.0%	12.8	58.4%	71.4%	33.8	75.9%	81.9%
li	1.64	0.7%	1.1%	4.13	1.3%	2.8%	13.0	1.4%	6.6%
ijpeg	1.35	19.7%	16.5%	2.99	14.5%	23.4%	9.75	20.4%	31.9%
perl	0.30	9.0%	0.3%	4.95	16.6%	8.9%	9.90	25.7%	17.6%
vortex	6.17	78.7%	38.7%	7.93	70.6%	47.1%	26.3	82.4%	49.1%
tomcatv	1.62	3.4%	0.6%	2.20	4.6%	1.5%	3.83	6.1%	2.7%
swim	6.35	10.6%	24.4%	21.0	24.8%	46.1%	22.0	20.2%	51.9%
mgrid	3.96	6.6%	77.5%	4.11	6.6%	78.9%	14.0	7.7%	89.6%
applu	4.93	6.3%	7.1%	4.99	6.6%	7.4%	19.5	30.6%	52.4%
apsi	4.50	20.9%	19.7%	5.28	23.8%	24.9%	12.5	20.6%	32.8%
wave5	13.6	14.1%	55.0%	21.7	16.9%	63.4%	48.2	16.6%	68.9%

```
    do {
         *(htab_p-16) = -1;
         *(htab_p-15) = -1;
         *(htab_p-14) = -1;
         *(htab_p-13) = -1;
         *(htab_p-12) = -1;
         *(htab_p-11) = -1;
         *(htab_p-10) = -1;
         *(htab_p-9) = -1;
         *(htab_p-8] = -1;
         *(htab_p-7) = -1;
         *(htab_p-6) = -1;
         *(htab_p-5) = -1;
         *(htab_p-4) = -1;
         *(htab_p-3) = -1;
         *(htab_p-2) = -1;
         *(htab_p-1) = -1;
          htab_p -= 16;
    } while ((i -= 16) >= 0);
```

FIGURE 9. ***compress:*** **Because these silent stores span across multiple cache lines, they are often critical.**

Because no non-silent stores can occur to a line if it is to avoid being written back, the greatest potential for writeback reduction exists when multiple silent stores occur to the same cache line. In other words, if a segment of code silently stores data that consumes an entire line, there is no room in that line for other data that, when stored to non-silently, can cause a writeback. Such a case is most prevalent inside a loop body when a store continually increments its address by a fixed offset (different location and different/same value). When the cache line is written back, every block has been silently stored to and the writeback can be avoided. A similar example of this occurs in *compress* (Figure 9) when the *cl_hash* function clears every entry in a large hash table resident in contiguous memory. Because many of the hash entries remain unused (and already contain the initialization value -1), most of the stores are silent. Each time this occurs for every entry in a line, a writeback can be removed, leading to over 76,000 removable writebacks from squashing only these silent stores (a 19% reduction in total writebacks for this benchmark).

Another situation in which writebacks can be removed occurs when multiple fields of a structure are silently stored to at nearly the same time. Because structure fields exhibit a great degree of spatial locality, it is likely that the silent stores occur within the same cache line and thus the writeback can be eliminated. The *makesim* function found in *m88ksim* illustrates this (Figure 10). In many cases the address pointed to by *opcode* and the results of the table lookup (*tblptr*) are identical to earlier invocations of *makesim* and the stores to *opcode*'s fields are silent. Additionally, because the size of the *IR_FIELDS* struct is the same as the simulated line size (32B), these stores are likely to fall within the same line, leading to over 1,000 removable writebacks if squashed.

To summarize, the issue of whether or not a silent store is critical depends on several factors, including the temporal locality of the address, spatial and temporal locality within a cacheline, cache size and configuration, and general program behavior.

5 .0 SVL Statistics

In this section we present data gathered in order to provide differentiating characteristics of silent stores. Much of the data is not machine model dependent, but occasionally the details of the memory hierarchy are relevant. Unless stated otherwise, the L1-I and L1-D caches are 32KB, 32-byte lines, and 2,4-way associative respectively. The L2 cache is unified, 256KB, 64-byte lines, and 4-way associative.

```
void makesim(unsigned int instr, struct IR_FIELDS *opcode){
  register INSTAB *tblptr;
  if(!(tblptr = lookupdisasm(instr & classify(instr))))
          tblptr = &simdata;
  opcode->op = tblptr->flgs.op;
  opcode->dest = uext(instr,tblptr->op1.offset,tblptr-op1.width);
  opcode->src1 = uext(instr, tblptr->op2.offset,tblptr->op2.width);
  opcode->p = tblptr;
  ......
}
```

FIGURE 10. ***m88ksim:*** **Critical silent stores often occur when stores of multiple structure fields are silent.**

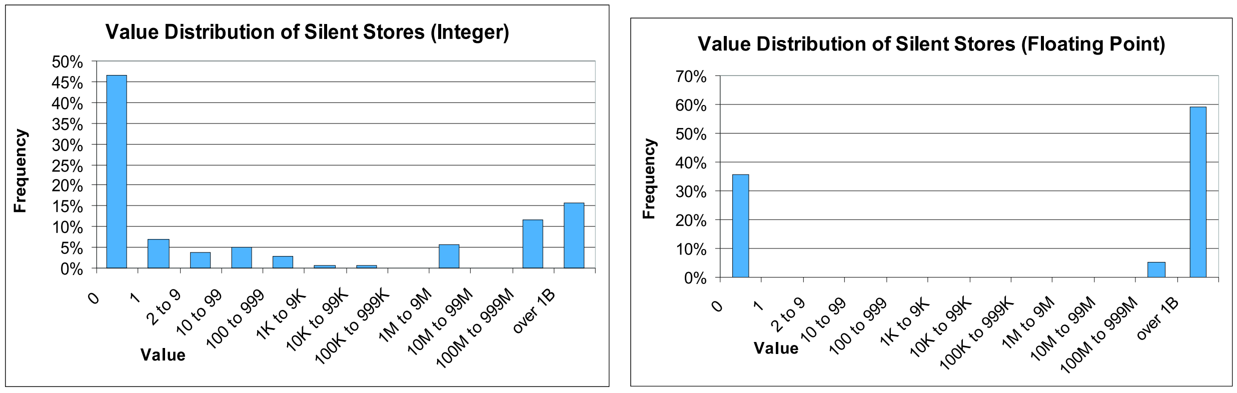


FIGURE 11. Unsigned values of silent stores in SPEC-95.

5.1 Value Distribution of SVL

Given that silent stores represent a significant portion of total program stores, which stored values have a tendency to be silent? That is, which values are likely to be stored to a memory location when they are already at that location? For each dynamic silent store, we recorded the value written, and tabulated the number of times we saw values within a given range. Figure 11 plots the results separated by floating point and integer stores within the SPECINT-95 and SPECFP-95 benchmarks (x-axis is a log scale). As perhaps expected, integer and floating point zero occurs very often. Not only is zero stored frequently in most programs (as an explicit value and as a null pointer), but uninitialized memory locations often (but not necessarily) already contain the value zero. We would expect this to be even more true in languages whose semantics mandate that memory be initialized to zero upon allocation (such as Java). For integer values, frequency quickly decreases as the value increases, until we reach values in excess of 1e6. At this point, we see the storage of address values (for the text segment, heap, and stack, respectively) which also contribute many silent stores. For the floating point results, we see that the majority of values are in excess of 1e9, most likely due to the large possible range of floating point values.

5.2 Frequency of Execution Related to SVL

Figure 12 examines a store's probability of being silent as a function of how many times it is dynamically executed. Each marker on the graph represents a single static store across all benchmarks (the integer figure has many more data points because its benchmarks generally have many more static stores than the floating point benchmarks). As the graph shows, the entire range of dynamic execution count leads to varying probabilities of a silent store. Nevertheless, definite trends can still be observed. In general, stores that are not executed many times have a decreased probability of being silent (based on the numerous stores in the lower left area of the integer graph). More importantly, many frequently executed stores still have a high tendency to be silent (because a non-trivial amount of stores exist in the upper right area of the graphs). Because these stores execute often and are silent a large percentage of the time, removing them will eliminate a significant portion of silent stores in a program. Thus efforts to remove silent stores should be initially directed toward stores that exhibit this type of behavior.

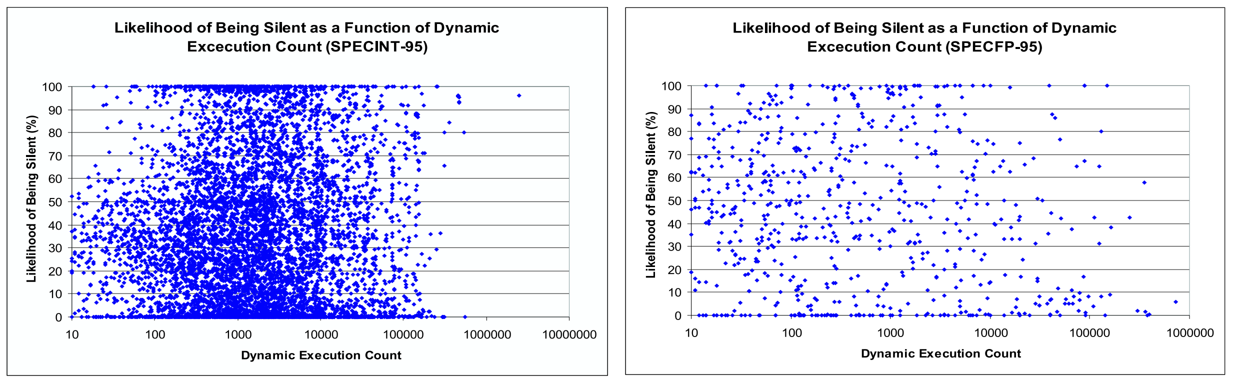


FIGURE 12. Probability of a store being silent as a function of dynamic execution count.

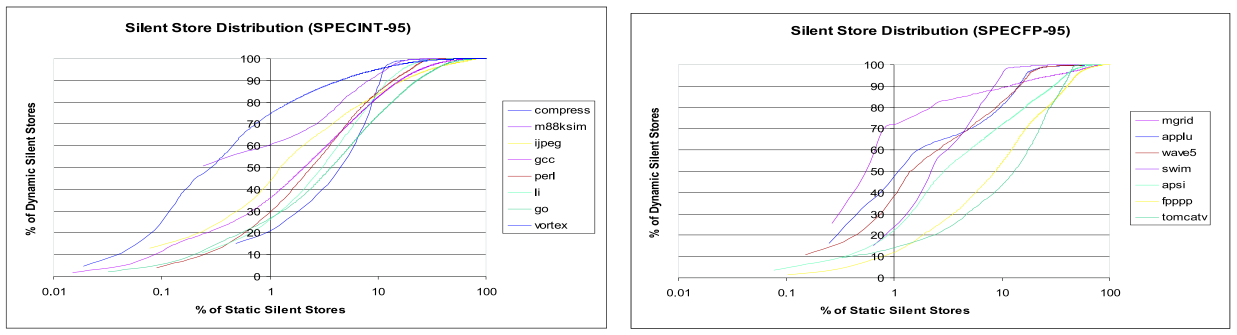


FIGURE 13. Total silent stores as a percentage of static silent stores (NOTE: the benchmark order in the legend matches the order when the x-axis is at 1%).

Figure 13 presents an alternate view of this data. It graphs the contribution of additional static silent stores (in order of decreasing number of total dynamic silent stores) toward the total number of dynamic silent stores in each benchmark. Note the log scale on both axes. In all of the integer cases, less than 25% of all static silent stores contribute over 90% of the total silent stores dynamically executed. The floating point benchmarks exhibit a similar, though not as dramatic, distribution. In certain benchmarks (such as *mgrid* and *compress*), 1% of static silent stores contribute over 70% of dynamic silent stores. Again, this uneven distribution of silent store contribution across a variety of programs reveals that elimination (or at least modification) of even only a few static stores can have a significant impact on silent store reduction.

5.3 Memory Hierarchy Related to SVL

Table 2 shows the percentage of stores that are silent as a function of where the store operation hits in the memory hierarchy. It is worthy to note that in this data, we assume a write-allocating cache, i.e. on a store (or equivalently, store-verify) miss, the cacheline containing the destination store is brought into the L1 cache before the actual store (store-verify) occurs. However, the data in Table 2 shows the original location in which the store (store-verify) hits. Examining the data in Table 2 leads to some interesting inferences about the possible program behavior that dictates a silent store. It is shown that for many cases, a store hitting in a location deeper in the memory hierarchy is more likely to be silent. Even for most of the exceptions to the above statement, we found (by a weighted combination of L2 and memory stores) that the combination of L2 and memory stores is more silent than the L1 stores (in *perl* and *tomcatv*, there are relatively few L2 and memory hits, mostly due to cold misses, making the statistics unreliable).

Why could this phenomenon occur? We believe a possible explanation is in the behavior of compilers. Although a compiler has extensive knowledge of the source code at compile time, in reality, in order for compilation to be realistically tractable, the compiler must look at optimizations and code generation at a local level.

Hence, it tends to generate fewer silent stores (as a percentage) for local code--as evidenced by the lower L1 cache percentages. As the locality of references the compiler looks at becomes more broad, it cannot analyze them concurrently and therefore does not do as well--evidenced by the higher L2 and memory percentages. However, this is not a complete explanation, as we've shown in Section 5.2 that highly silent stores can execute often, and are thus likely to be resident in cache levels closer to the processor.

These results can also be explained in terms of working set size. In the case of *compress* (Figure 14), a stream of input data is written into a buffer (*new_count*), compressed into a second buffer (*comp_count*), and finally decompressed into a third (*new_count*). The process is then repeated 25 times, each iteration on a slightly larger, but mostly unchanged input set. When a new iteration begins, the input data is written over the old input buffer, and because the two data sets are virtually identical, most of the stores are silent. However the simulated cache is not large enough for all three buffers, and generating the second and third cause the original input buffer to be ejected from the cache. Thus many of compress's silent stores occur in deeper levels of the memory hierarchy.

Table 2: Memory hierarchy silent store statistics.

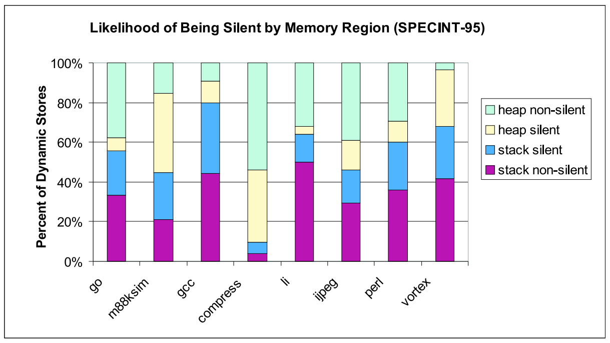
SPEC Benchmark	% of silent L1 cache hits	% of silent L2 cache hits	% of silent Memory hits
go	29%	41%	31%
m88ksim	63%	74%	88%
gcc	47%	41%	50%
compress	40%	86%	91%
li	18%	10%	74%
ijpeg	32%	42%	47%
perl	34%	16%	12%
vortex	54%	87%	94%
tomcatv	34%	5%	11%
swim	24%	24%	24%
mgrid	7%	11%	12%
applu	31%	62%	90%
apsi	25%	22%	42%
fpppp	15%	26%	21%
wave5	22%	17%	34%

```
fill_text_buffer(count, start_char, orig_text_buffer);
for (i = 1; i <= 25; i++){
    new_count=add_line(orig_text_buffer, count, i, start_char);
    count=new_count;
    oper=COMPRESS;
    printf("The starting size is: %d\n", count);
    comp_count=spec_select_action(orig_text_buffer,count,oper,comp_text_buffer);
    printf("The compressed size is: %d\n", comp_count);
    oper=UNCOMPRESS;
    new_count=spec_select_action(comp_text_buffer,comp_count,oper,new_text_buffer);
    printf("The compressed/uncompressed size is: %d\n", new_count);
    compare_buffer(orig_text_buffer,count,new_text_buffer, new_count);
}
```

FIGURE 14. Main loop in *compress*.

5.4 Stack/Heap SVL

Most architectures have a distinct way of dividing memory areas into a stack and heap region. We examined whether a store was more likely to be silent depending on which portion of memory it was to, hoping to gain some insight as to whether function parameters or some other "local" variables were more likely to be silent than heap-allocated data. In our simulator, the stack begins at the largest address of memory, and the heap begins at the smallest. We counted stack and heap references by choosing an address threshold (1 billion) and counting separate silent store statistics for instructions that generated effective addresses above and below this threshold. The results are shown in Figure 15.



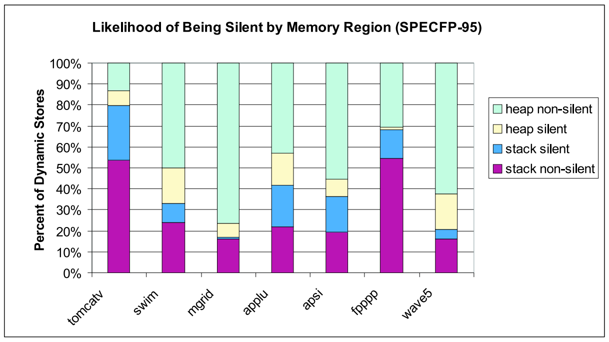


FIGURE 15. Silent stores categorized by memory region.

In general, little variance exists between benchmarks in how likely a store to the stack is of being silent. This suggests that there are specific attributes of how the stack is used by architectures and compilers that dictate silent stores (for example, callee-saving register values, saving return addresses, and parameter passing between functions). Because the percent of stack stores that are silent is nontrivial in most of the benchmarks (typically between 25%-50% with the exception of *mgrid*, and strongly correlated to the overall utilization of the stack in a given benchmark), it seems likely that modifications in the way that the compiler manipulates stack frames could eliminate a substantial number of silent stores. Conversely, the benchmarks exhibit high variability in the percent of heap stores that are silent. Sometimes there are far more non-silent than silent heap stores, while other times the reverse is true. This can be explained by the fact that virtually all heap accesses are programmer directed--the compiler does not transparently manipulate it as it does to the stack. Thus the notion of whether a heap store is silent is largely algorithmic, and so varies greatly between programs. We also see from Figure 15 that the breakdown of total stack and heap stores differs across the benchmarks. Control-intensive programs with many function calls (such as *gcc*) have many stack stores due to their large number of stack frame allocations, while programs that operate on large heap-allocated data sets (such as *compress* and many floating point applications) do not. Additionally, many data structures can be stack or heap-allocated, and the decision of which differs among programmers and applications.

5.5 Compiler Optimizations

The amount of silent stores present can be affected by compiler optimizations. Because compilers do not have an infinite optimization window, they frequently cannot determine which instruction last modified a particular memory location. It seems plausible that if the compiler is allowed to optimize more aggressively, it might be able to identify many silent stores. Table 4 shows silent store statistics with SimpleScalar on binaries compiled with gcc at optimization of "-O0" and optimization "-O3". It breaks down the reduction of total stores into stores that are silent and those that are not (the percent of stores silent for "-O3" optimization is given in Table 1). In all of the benchmarks except *li*, the compiler was able to eliminate more non-silent than silent stores. Thus the overall percentage of silent stores actually increased when optimization was enabled. Even though gcc may not apply optimizations as aggressively as other compilers, it provides the only method for compilation to SimpleScalar PISA binaries. However, even with an industrial compiler, such as IBM's AIX optimizing C compiler, the percentage of silent stores is non-trivial (in fact, it is higher overall than the results here, as seen in [7]).

Why is a compiler able to eliminate fewer stores that are more likely to be silent? We believe this occurs because many silent stores are largely algorithmic in nature. As we increase the compilation optimization level, we remove many artifacts of “language overhead” that the compiler already knows how to analyze. Take function inlining as a simple hypothetical example--in this case, many stores (silent and non-silent) due to register saving conventions, etc. (as outlined in Section 4.2) will be removed. This leaves relatively more algorithm-induced stores, which the compiler is not able to analyze as easily, and hence the percentage of silent stores increases. A similar argument may also be made for other common optimizations, but we do not explore these optimizations individually in this work. Also, optimization-independent architectural considerations that a compiler adheres to (such as callee-saving of register values on the stack discussed in Section 4.2), or language requirements (such the volatile keyword in C) may introduce additional silent stores.

In order to statically remove a silent store from a program, the compiler must prove that it is silent for every execution under any input data set (as opposed to dynamic squashing, which requires that a store is silent for only one execution). Although it is certainly possible to optimize stores that are not silent all of the time (for example, by adding a store verify in software), the semantic analysis required to determine when stores are likely to be silent is probably too great without the aid of profiling information. Furthermore, even though the optimization window may be increased to include more instructions, we showed in Section 5.3 that enough time often passes between two stores such that the silent value has left the caches and is resident in main memory. In this case it is unlikely that even a moderately larger window will enable the compiler to see both stores. However, it may be possible for a compiler having algorithms considering store value locality to remove silent stores. We do not explore any such algorithms in this work, leaving this as a topic of future investigation.

Table 3: The effect of optimization on silent stores.

	No optimization (-O0)			Optimization (-O3)	
SPEC Benchmark	Silent Percent	Non-silent Stores	Silent Stores	Non-silent Stores (% reduction)	Silent Stores (% reduction)
go	27%	9.2M	3.5M	4.0M (57%)	1.7M (51%)
m88ksim	42%	10.9M	7.9M	3.3M (70%)	5.6M (29%)
gcc	43%	6.7M	4.9M	4.3M (36%)	3.8M (22%)
compress	38%	6.0M	3.7M	3.5M (42%)	2.5M (32%)
li	22%	8.3M	2.5M	6.0M (28%)	1.3M (48%)
ijpeg	19%	13.0M	3.0M	4.9M (62%)	1.9M (37%)
perl	35%	5.7M	3.0M	4.7M (18%)	2.5M (17%)
vortex	48%	10.7M	9.7M	7.4M (31%)	9.0M (7%)

6 .0 Conclusion

This work affirms that store value locality occurs with significant frequency in real applications. It revisits the notion of silent stores and the fact that, because they do not change the state of the processor, they can safely be removed from program execution. Such removal is desirable and is motivated through potential microarchitectural simplifications such as reduced write-back buffers and simpler CAMs. Source code was analyzed in order to determine common underlying causes of silent stores. This work demonstrates that silent stores occur in all levels of program execution and compiler optimization. Additionally, it shows that frequently occurring stores are highly likely to be silent. The notion of critical silent stores is also introduced, followed by a description on how to identify them. Removing the small subset of silent stores that are critical is sufficient for removing all of the avoidable cache writebacks.

As shown in several of the examples, the fact that many stores are silent creates the opportunity for removing a significant fraction of all the instructions in the dynamic instruction stream, since many of those instructions are merely computing the redundant results being stored by the silent stores. Investigation of static and dynamic techniques for removing such redundant computation is left to future work. In summary, this work explores and illuminates several aspects of store value locality and presents opportunities for further investigation.

References

[1] S. Adve and K. Gharachorloo. “Shared Memory Consistency Models: A Tutorial.” DECWRL Technical Report, September 1995.
[2] A. Appel and M. Ginsburg. “Modern Compiler Implementation in C.” Cambridge University Press, Cambride, NY, 1998.
[3] D. Burger and T. M. Austin. “The SimpleScalar Tool Set Version 2.0”. University of Wisconsin Computer Sciences Technical Report #1342. June 1997.
[4] B. Calder, P. Feller, and A. Eustace. “Value Profiling.” In Proc. of MICRO-30, pages 259-269, December 1997.
[5] S. P. Harbison. “An Architectural Alternative to Optimizing Compilers.” In Proc. of ASPLOS, pages 57-65, March 1982.
[6] J.L. Hennessy and D.A. Patterson. “Computer Architecture: A Quantitative Approach.” Morgan Kauffman Publishers, Inc., San Mateo, CA, 1990.
[7] K. M. Lepak and M. H. Lipasti. “On the Value Locality of Store Instructions.” To appear in ISCA-2000, June, 2000.
[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. “Value locality and load value prediction.” In Proceedings of ASPLOS-VII, October 1996.
[9] M. H. Lipasti and J. P. Shen. “Exceeding the dataflow limit via value prediction.” In Proceedings of MICRO-29, December 1996.
[10]A. Mendelson and F. Gabbay. “Speculative execution based on value prediction.” Technical report, Technion, 1997. (http://www-ee.technion.ac.il/ fredg)
[11]C. Molina and A. Gonzalez and J. Tubella. “Reducing Memory Traffic Via Redundant Store Instructions.” Proc. of Conf. on High Perf. Computing and Networking. April 1999.
[12]S. E. Richardson. “Exploiting Trivial and Redundant Computation.” In Proc. of the 11th Symp. on Computer Arithmetic, pages 220-227, July 1993.
[13]A. Silberschatz and P. B. Galvin. “Operating Systems Concepts.” Addison-Wesley Longman, Inc., Reading, MA, 1998.
[14]http://simos.stanford.edu
[15]A. Sodani and G. Sohi. “Dynamic Instruction Reuse.” In ISCA-1997, June 1997.
[16]http://www.spec.org