Using a Swap Instruction
to Coalesce Loads and Stores

Apan Qasem, David Whalley, Xin Yuan, and Robert van Engelen

Department of Computer Science, Florida State University
Tallahassee, FL 32306-4530, U.S.A.
e-mail: {qasem,whalley,xyuan,engelen}@cs.fsu.edu, phone: (850) 644-3506

Abstract. A swap instruction, which exchanges a value in memory with
a value of a register, is available on many architectures. The primary
application of a swap instruction has been for process synchronization.
As an experiment we wished to see how often a swap instruction can be
used to coalesce loads and stores to improve the performance of a variety
of applications. The results show that both the number of accesses to the
memory system (data cache) and the number of executed instructions
are reduced.

1 INTRODUCTION

An instruction that exchanges a value in memory with a value in a register
has been used on a variety of machines. The primary purpose for these swap
instructions is to provide an atomic operation for reading from and writing
to memory, which has been used to construct mutual-exclusion mechanisms in
software for process synchronization. In fact, there are other forms of hardware
instructions that have been used to support mutual exclusion, which include
the classic test-and-set instruction. We thought it would be interesting to see if
a swap instruction could be exploited in a more conventional manner. In this
paper we show that a swap instruction can also be used by a low-level code-
improving transformation to coalesce loads and stores into a single instruction,
which results in a reduction of memory references and executed instructions.

A swap instruction described in this paper exchanges a value in memory with
a value in a register. This is illustrated in Fig. 1, which depicts a load instruction,
a store instruction, and a swap instruction using an RTL (register transfer list)
notation. Each assignment in an RTL represents an effect on the machine. The
list of effects within a single RTL are accomplished in parallel. Thus, the swap
instruction is essentially a load and store accomplished in parallel.

\[
\begin{align*}
    r[2] &= M[x] ; &
    M[x] &= r[2] ; &
    r[2] &= M[x] ; &
    M[x] &= r[2] ; \\
\end{align*}
\]

(a) Load Instruction (b) Store Instruction (c) Swap Instruction

Fig. 1. Contrasting the Effects of Load, Store, and Swap Instructions
for \( j = n-1; j > 1; j-- \) {
\[
d[j] = d[j-1] - dd[j];
\]
}\)

(b) Loop after Unrolling

Fig. 2. Unrolling a Loop to Provide an Opportunity to Exploit a Swap Instruction

2 OPPORTUNITIES FOR EXPLOITING A SWAP

A swap instruction can potentially be exploited when a load is followed by a store to the same memory address and the value stored is not computed using the value that was loaded. We investigated how often this situation occurs and we have found many direct opportunities in a number of applications. The most common situation is when the values of two variables are exchanged. However, there are also opportunities for exploiting a swap instruction after other code-improving transformations have been performed. It would appear in the code segment of Fig. 2(a) that there is no opportunity for exploiting a swap instruction. However, consider Fig. 2(b) which shows the loop unrolled by a factor of two. Now the value loaded from \( d[j-1] \) in the first assignment statement in the loop is updated in the second assignment statement and the value computed in the first assignment is not used to compute the value stored in the second assignment.

Sometimes apparent opportunities at the source code level for exploiting a swap instruction are not available after other code-improving transformations have been applied. Many code-improving transformations either eliminate (e.g., register allocation) or move (e.g., loop-invariant code motion) memory references. Coalescing loads and stores into swap instructions should only be performed after all other code-improving transformations that can affect the memory references have been applied. Fig. 3(a) shows an exchange of values after the two values are compared in an \( if \) statement. Fig. 3(b) shows a possible translation of this code segment to machine instructions. Due to common subexpression elimination, the loads of \( x \) and \( y \) in the block following the branch have been deleted in Fig. 3(c). Thus, the swap instruction cannot be exploited within that block.

\[
\text{if} \ (x > y) \{ \\
\quad t = x; \\
\quad x = y; \\
\quad y = t; \\
\}
\]
(a) Exchange of Values in \( x \) and \( y \) at the Source Code Level

\[
\begin{align*}
\text{r[1]} &= M[x]; \\
\text{r[2]} &= M[y]; \\
\text{IC} &= \text{r[1]} \, ? \, \text{r[2]}; \\
\text{PC} &= \text{IC} \, <= \, 0 \, , \, \text{L5}; \\
\text{r[1]} &= M[x]; \\
\text{r[2]} &= M[y]; \\
\text{M}[x] &= \text{r[2]}; \\
\text{M}[y] &= \text{r[1]}; \\
\end{align*}
\]
(b) Loads are Initially Performed in the Exchange of Values of \( x \) and \( y \)

\[
\begin{align*}
\text{r[1]} &= M[x]; \\
\text{r[2]} &= M[y]; \\
\text{IC} &= \text{r[1]} \, ? \, \text{r[2]}; \\
\text{PC} &= \text{IC} \, <= \, 0 \, , \, \text{L5}; \\
\text{M}[x] &= \text{r[2]}; \\
\text{M}[y] &= \text{r[1]}; \\
\end{align*}
\]
(c) Loads Are Deleted in the Exchange of Values Due to Common Subexpression Elimination

Fig. 3. Example Depicting Load Instructions Being Deleted
Fig. 4. Example of Exchanging the Values of Two Variables

3 A CODE-IMPROVING TRANSFORMATION TO EXPLOIT THE SWAP INSTRUCTION

Fig. 4(a), shows an exchange of the values of two variables, x and y, at the source code level. Fig. 4(b) shows similar code at the SPARC machine code level, which is represented in RTLs. The variable t has been allocated to register r[1]. Register r[2] is used to hold the temporary value loaded from y and stored in x. At this point a swap could be used to coalesce the load and store of x or the load and store of y. Fig. 4(c) shows the RTLs after coalescing the load and store of x. One should note that r[1] is no longer used since its live range has been renamed to r[2]. Due to the renaming of the register, the register pressure at this point in the program flow graph has been reduced by one. Reducing the register pressure can sometimes enable other code-improving transformations that require an available register to be applied. Note that the decision to coalesce the load and store of x prevents the coalescing of the load and store of y.

The transformation to coalesce a load and a store into a swap instruction was accomplished using an algorithm described in detail elsewhere [4]. The algorithm finds a load followed by a store to the same address and coalesces the two memory references together into a single swap instruction if a number conditions are met. Due to space constraints, we only present a few of the conditions.

The instruction containing the first use of the register assigned by the load has to occur after the last reference to the register to be stored. For example, consider the example in Fig. 5(a). A use of r[a] appears before the last reference to r[b] before the store instruction, which prevents the load and store from being coalesced. Fig. 5(b) shows that our compiler is able to reschedule the instructions between the load and the store to meet this condition. Now the load can be moved immediately before the store, as shown in Fig. 5(c). Once the load and store are contiguous, the two instructions can be coalesced. Fig. 5(d) shows the code sequence after the load and store have been deleted, the swap instruction has been inserted, and r[a] has been renamed to r[b].

4 RESULTS

Table 1 describes the numerous benchmarks and applications that we used to evaluate the impact of applying the code-improving transformation to coalesce loads and stores into a swap instruction. The code-improving transformation was
implemented in the *vpo* compiler [1]. *Vpo* is a compiler backend that is part of the *zephyr* system, which is supported by the National Compiler Infrastructure project. The programs depicted in boldface were directly obtained from the Numerical Recipes in C text [3]. The code in many of these benchmarks are used as utilities in a variety of programs. Thus, coalescing loads and stores into swaps can be performed on a diverse set of applications.

Table 2 depicts the results that were obtained on the test programs for coalescing loads and stores into swap instructions. We unrolled several loops in these programs by an unroll factor of two to provide opportunities for coalescing a load and a store across the original iterations of the loop. In these cases, the *Not Coalesced* column includes the unrolling of these loops to provide a fair comparison. The results show decreases in the number of instructions executed and memory references performed for a wide variety of applications. The amount of the decrease varied depending on the execution frequency of the load and store instructions that were coalesced. As expected the use of a swap instruction did not decrease the number of data cache misses.

### Table 1. Test Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>banddec</td>
<td>constructs an LU decomposition of a sparse representation of a band diagonal matrix</td>
</tr>
<tr>
<td>bubblesort</td>
<td>sorts an integer array in ascending order using a bubble sort</td>
</tr>
<tr>
<td>chebpc</td>
<td>polynomial approximation from Chebyshev coefficients</td>
</tr>
<tr>
<td>elmhes</td>
<td>reduces an $N \times N$ matrix to Hessenberg form</td>
</tr>
<tr>
<td>fft</td>
<td>fast fourier transform</td>
</tr>
<tr>
<td>gaussj</td>
<td>solves linear equations using Gauss-Jordan elimination</td>
</tr>
<tr>
<td>indexx</td>
<td>cal. indices for the array such that the indices are in ascending order</td>
</tr>
<tr>
<td>ludcmp</td>
<td>performs LU decomposition of an $N \times N$ matrix</td>
</tr>
<tr>
<td>mmid</td>
<td>modified midpoint method</td>
</tr>
<tr>
<td>predic</td>
<td>performs linear prediction of a set of data points</td>
</tr>
<tr>
<td>select</td>
<td>finds the root of a function using the false position method</td>
</tr>
<tr>
<td>thresh</td>
<td>adjusts an image according to a threshold value</td>
</tr>
<tr>
<td>transpose</td>
<td>transposes a matrix</td>
</tr>
<tr>
<td>traverse</td>
<td>binary tree traversal without a stack</td>
</tr>
<tr>
<td>tsp</td>
<td>traveling salesman problem</td>
</tr>
</tbody>
</table>

### 5 CONCLUSIONS

In this paper we have experimented with exploiting a swap instruction, which exchanges the values between a register and a location in memory. While a
Table 2. Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Instructions Executed</th>
<th>Memory References Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not Coalesced</td>
<td>Coalesced</td>
</tr>
<tr>
<td>...</td>
<td>69.189</td>
<td>68.459</td>
</tr>
<tr>
<td>...</td>
<td>2,439,005</td>
<td>2,376,705</td>
</tr>
<tr>
<td>...</td>
<td>7,531,984</td>
<td>7,029,900</td>
</tr>
<tr>
<td>...</td>
<td>18.927</td>
<td>18.644</td>
</tr>
<tr>
<td>...</td>
<td>4,176,112</td>
<td>4,148,112</td>
</tr>
<tr>
<td>...</td>
<td>27,143</td>
<td>26,756</td>
</tr>
<tr>
<td>...</td>
<td>70,322</td>
<td>68,676</td>
</tr>
<tr>
<td>...</td>
<td>10,521,952</td>
<td>10,439,152</td>
</tr>
<tr>
<td>...</td>
<td>267,563</td>
<td>258,554</td>
</tr>
<tr>
<td>...</td>
<td>40,827</td>
<td>39,927</td>
</tr>
<tr>
<td>...</td>
<td>81,117</td>
<td>80,116</td>
</tr>
<tr>
<td>...</td>
<td>19,039</td>
<td>19,463</td>
</tr>
<tr>
<td>...</td>
<td>7,958,909</td>
<td>7,661,706</td>
</tr>
<tr>
<td>...</td>
<td>42,883</td>
<td>37,993</td>
</tr>
<tr>
<td>...</td>
<td>94,159</td>
<td>91,099</td>
</tr>
<tr>
<td>...</td>
<td>64,294,814</td>
<td>63,950,122</td>
</tr>
<tr>
<td>...</td>
<td>6,103,402</td>
<td>6,019,616</td>
</tr>
</tbody>
</table>

swap instruction has traditionally only been used for process synchronization, we wished to determine if a swap instruction could be used to coalesce loads and stores. Different types of opportunities for exploiting the swap instruction were shown to be available. A number of issues related to implementing the coalescing transformations were described. The results show that this code-improving transformation could be applied on a variety of applications and benchmarks and reductions in the number of instructions executed and memory references performed were observed.

6 ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation grants EIA-9806525, CCR-9904943, CCR-0073482, and EIA-0072043.

References


