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 if a swap instruction can often be used to coalesce loads and stores in a variety of applications. We describe the analysis necessary to detect opportunities to exploit a swap and the transformation required to coalesce a load and a store into a swap instruction. The results show that both the number of accesses to the memory system (data cache) and the number of executed instructions are reduced. In addition, the transformation reduces the register pressure by one register at the point the swap instruction is used, which sometimes enables other code-improving transformations to be performed. Using a swap instruction can also allow additional ILP scheduling opportunities by eliminating structural hazards to memory by reducing the number of memory accesses performed.

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.

It is possible that a swap instruction can be efficiently integrated into a conventional RISC architecture.

```
r[2] = M[x];          M[x] = r[2];          r[2] = M[x]; M[x] = r[2];
```

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

Fig. 1. Contrasting the Effects of Load, Store, and Swap Instructions

1. It can be encoded using the same format as a load or a store since all three instructions reference a register and a memory address. Only additional opcodes are required to encode swap instructions.
2. Access to a data cache can potentially be performed efficiently for a swap instruction. A direct-mapped data cache can send the value to be loaded from memory to the processor for a load or a swap instruction in parallel with the tag check. This value will not be used by the processor if a tag mismatch is later discovered [7]. A data cache is not updated with the value to be stored by a store or a swap instruction until after the tag check [7]. Thus, a swap instruction could be performed as efficiently as a store instruction on a machine with a direct-mapped data cache. In fact, a swap instruction requires the same number of cycles in the pipeline as a store instruction on the MicroSPARC I [10]. One should note that a swap instruction will likely perform less efficiently when it is used for process synchronization on a multiprocessor machine since it requires a signal over the bus to prevent other accesses to memory.
3. It is possible that a main memory access could also be performed efficiently for a swap instruction. Reads to DRAM are destructive, meaning that the value read must be written back afterwards. A DRAM organization could be constructed where the value that is written back could differ from the value that was read and sent to the processor. Thus, a load and a store to a single word of main memory could occur in one main memory access cycle.

The remainder of this paper has the following organization. First, we introduce related work that allows multiple accesses to memory to occur simultaneously. Second, we describe a variety of different opportunities for exploiting a swap instruction that commonly appear in applications. Third, we present the conditions necessary for coalescing a load and store pair into a swap instruction. Fourth, we give the results of applying the code-improving transformation on a variety of applications. Finally, we state the conclusions of the paper.

2 RELATED WORK

There has been some related work that allows multiple accesses to the memory system to occur in a single cycle. Superscalar and VLIW machines have been developed where a wider datapath between the data cache and the processor has been used to allow multiple simultaneous accesses to the data cache. Likewise, a wider datapath has been implemented between the data cache and main memory to allow multiple simultaneous accesses to main memory through the use of memory banks. A significant amount of compiler research has been spent on trying to schedule instructions so that multiple independent memory accesses to different banks can be performed simultaneously [6].

Memory access coalescing is a code-improving transformation that groups multiple memory references to consecutive memory locations into a single larger memory reference. This transformation was accomplished by recognizing a contiguous access pattern for a memory reference across iterations of a loop, unrolling the loop, and rescheduling instructions so that multiple loads or stores could be coalesced [4].

Direct hardware support of multiple simultaneous memory accesses in the form of superscalar or VLIW architectures requires that these simultaneous memory accesses be independent in that they access different memory locations. Likewise, memory access coalescing requires that the coalesced loads or stores access contiguous (and different) memory locations. In contrast, the use of a swap instruction allows a store and a load to the same memory location to be coalesced together and performed simultaneously. In a manner similar to the memory access coalescing transformation, a load and a store are coalesced together and explicitly represented in a single instruction.

3 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. Consider the following code segment in Fig. 2 from an application that uses polynomial approximation from Chebyshev coefficients [9]. There is first a load of the `d[k]` array element followed by a store to the same element, where the store does not use the value that was loaded. We have found comparable code segments containing such a load followed by a store in other diverse applications, such as Gauss-Jordan elimination [9] and tree traversals [5].

```
...
sv = d[k];
d[k] = 2.0*d[k-1] - dd[k];
...
```

Fig. 2. Code Segment in Polynomial Approximation from Chebyshev Coefficients

A more common operation where a swap instruction can be exploited is when the values of two variables are exchanged. Consider Fig. 3(a), which depicts the exchange of the values in `x` and `y` at the source code level. Fig. 3(b) indicates that the load and store of `x` can be coalesced together. Likewise, Fig. 3(c) indicates that the load and store of `y` can also be coalesced together. However, we will discover in the next section that only a single pair of load and store instructions in an exchange of values between variables can be coalesced together.

```
t  =  x;
x  =  y;
y  =  t;
```

(a) Exchange of Values between x and y

```
t  =  x;
x  =  y;
y  =  t;
```

(b) The Load and Store of x Can Be Coalesced

```
t  =  x;
x  =  y;
y  =  t;
```

(c) The Load and Store of y Can Be Coalesced

Fig. 3. Example of Exchanging the Values of Two Variables

There are numerous applications where the values of two variables are exchanged. Various sorts of an array or list of values are obvious applications in which a swap instruction could be exploited. Some other applications requiring an explicit exchange of values between two variables include transposing a matrix, the traveling salesperson problem, solving linear algebraic equations, fast fourier transforms, and the integration of differential equations. The above list is only a small subset of the applications that require this basic operation.

There are also opportunities for exploiting a swap instruction after other code-improving transformations have been performed. Consider the code segment in Fig. 4(a) from an application that uses polynomial approximation from Chebyshev coefficients [9]. It would appear in this code segment that there is no opportunity for exploiting a swap instruction. However, consider the body of the loop executed across two iterations, which is shown in Fig. 4(b) after unrolling the loop by a factor of two. For simplicity, we are assuming in this example that the original loop iterated an even number of times. 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. We have found opportunities for exploiting a swap instruction across loop iterations by loop unrolling in a number of applications, which includes linear prediction, interpolation and extrapolation, and solution of linear algebraic equations.

```
for (j = n-1; j > 1; j--)
   d[j] = d[j-1]-dd[j];
```

(a) Original Loop

```
for (j = n-1; j > 1; j -= 2) {
   d[j] = d[j-1]-dd[j];
   d[j-1] = d[j-2]-dd[j-1];
}
```

(b) Loop after Unrolling

Fig. 4. Unrolling a Loop to Provide an Opportunity to Exploit a Swap Instruction

Loop unrolling increases the opportunities for exploiting more swap instructions in a loop by coalescing loads and stores across loop iterations. When the distance vector of the anti-dependencies of a pair of array references in a one-dimensional loop is some positive integer *Distance*, and we assume that the loop unroll factor is a multiple of *Distance*, that is

$$Loop\ Unroll\ Factor = k \cdot Distance \tag{1}$$

```
for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      if (p[i][j] <= t)
          p[i][j] = 1;
      else
          p[i][j] = 0;
```

Fig. 5. Speculative Use of a Swap Instruction

for some positive integer constant k, then it can be shown that

$$Static\ Memory\ References\ Saved = \left\lfloor \frac{k}{2} \right\rfloor \cdot Distance \tag{2}$$

For example in Fig. 4, we have $Distance = 1$ and k equals the loop unroll factor. Thus, one memory reference is saved each time the loop is unrolled twice.

As the loop unroll factor increases, more memory references can be saved by coalescing loads and stores into swap instructions. However, the code size increase due to increased loop unrolling may not justify the reduction in memory references performed. Of course, loop unrolling provides additional benefits, such as reduced loop overhead and better opportunities for scheduling instructions.

Finally, we have also discovered opportunities for speculatively exploiting a swap instruction across basic blocks. Consider the code segment in Fig. 5, which assigns values to an image according to a specified threshold [8]. `p[i][j]` is loaded in one block and a value is assigned to `p[i][j]` in both of its successor blocks. The load of `p[i][j]` and the store from the assignment to `p[i][j]` in the `then` or `else` portions of the `if` statement can be coalesced into a swap instruction since the value loaded is not used to compute the value stored and the memory location is updated in both paths. The store operation can be speculatively performed as part of a swap instruction in the block containing the load. We have found that stores can be performed speculatively in a number of other image processing applications, which include clipping and arithmetic operations.

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 memory references (e.g. register allocation) or move memory references (e.g. loop-invariant code motion). 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. 6(a) shows an exchange of values after the two values are compared in an `if` statement. Fig. 6(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. 6(c). Thus, the swap instruction cannot be exploited within that block. This example illustrates why the swap instruction should be performed late in the compilation process when the actual loads and stores that will remain in the generated code are known.

`if (x > y) {`
`    t = x;`
`    x = y;`
`    y = t;`
`}`
	`r[1] = M[x];`
	`r[2] = M[y];`
	`IC = r[1] ? r[2];`
	`PC = IC <= 0, L5;`
	`r[1] = M[x];`
	`r[2] = M[y];`
	`M[x] = r[2];`
	`M[y] = r[1];`
		`r[1] = M[x];`
		`r[2] = M[y];`
		`IC = r[1] ? r[2];`
		`PC = IC <= 0, L5;`
		`M[x] = r[2];`
		`M[y] = r[1];`

(a) Exchange of Values in x and y at the Source Code Level	**(b) Loads are Initially Performed in the Exchange of Values of x and y**	**(c) Loads Are Deleted in the Exchange of Values Due to Common Subexpression Elimination**

Fig. 6. Example Depicting Load Instructions Being Deleted

4 A CODE-IMPROVING TRANSFORMATION TO EXPLOIT THE SWAP INSTRUCTION

Fig. 7(a) illustrates the general form of a load followed by a store that can be coalesced. The memory reference is to the same variable or location and the register loaded (`r[a]`) and register stored (`r[b]`) differ. Fig. 7(b) depicts the swap instruction that represents the coalesced load and store. Note that the register loaded has been renamed from `r[a]` to `r[b]`. This renaming is required since the swap instruction has to store from and load into the same register.

```
r[a] = M[v];
...
M[v] = r[b];
```

(a) Load Followed by a Store

```
r[b] = M[v]; M[v] = r[b];
```

(b) Coalesced Load and Store

Fig. 7. Simple Example of Coalescing a Load and Store into a Swap Instruction

Fig. 8(a), like Fig. 3(a), shows an exchange of the values of two variables, `x` and `y`, at the source code level. Fig. 8(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. 8(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 code-improving transformation to coalesce a load and a store into a swap instruction was accomplished using the algorithm in Fig. 9. Due to space

(a) Exchange of Values in x and y at the Source Code Level	(b) Exchange of Values in x and y at the Machine Code Level	(c) After Coalescing the Load and Store of x

`t = x;`
`x = y;`
`y = t;`
	`r[1] = M[x];`
	`r[2] = M[y];`
	`M[x] = r[2];`
	`M[y] = r[1];`
		`r[2] = M[y];`
		`M[x] = r[2]; r[2] = M[x];`
		`M[y] = r[2];`

Fig. 8. Example of Exchanging the Values of Two Variables

```
FOR B = each block in function DO
   FOR LD = each instruction in B DO
      IF LD is a load AND Find_Matching_Store(LD, B, LD→next, ST)
        AND Meet_Swap_Conds(LD, ST) THEN
        SWAP = Create("%s=M[%s];M[%s]=%s;",ST→r[b], LD→load_addr,
                      LD→load_addr, ST→r[b]);
        Insert SWAP after last reference to r[b] and before the first use of r[a];
        Replace uses of L→r[a] with S→r[b] until L→r[a] dies;
        Delete LD and ST;

BOOL Find_Matching_Store(LD, B, FIRST, ST) {
FOR ST = FIRST to B→last DO
   IF ST is a store THEN
      IF ST→store_addr == LD→load_addr THEN
        RETURN TRUE;
      IF ST→store_addr != LD→load_addr THEN
        CONTINUE;
      IF cannot determine if the two addresses are same or different THEN
        RETURN FALSE;
IF FIRST == B→first THEN
   RETURN FALSE;
FOR S = each successor of B DO
   IF !Find_Matching_Store(LD, S, S→first, ST) THEN
      RETURN FALSE;
RETURN TRUE;
}
```

Fig. 9. Algorithm for Coalescing a Load and a Store into a Swap Instruction

constraints, many details had to be left out. 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 the following conditions are met.

(1) The store must follow the load within the same block or consecutively executed blocks. If the load is in a separate block, then we must guarantee that a store to the same location occurs in each successor, which is the case in Fig. 5.

(2) The addresses of the memory references in the load and store instructions have to be the same. Fig. 10(a) shows a load using the address in register `r[2]` and a store using the address in `r[4]`. The compiler must ensure that the value in `r[2]` is the same as that in `r[4]`. This process of checking that two addresses are equivalent is complicated due to our code-improving transformation being performed late in the compilation process. We addressed this problem by associating each memory reference with a symbolic address early in the compilation process and tracked the movement of memory references through a variety of code-improving transformations.

```
r[2] = r[3] << 2;
r[2] = r[2] + _a;
r[5] = M[r[2]];
...
r[4] = r[3] << 2;
r[4] = r[4] + _a;
M[r[4]] = r[6];
```

(a) Same Addresses after Expanding the Expressions

```
r[a] = M[v];
...
M[r[c]] = r[d];
...
M[v] = r[b];
```

(b) Load and Store to the Same Variable with an Intervening Store

```
r[a] = M[v];

r[b] = r[a];

M[v] = r[b];
```

(c) A Store That Is Dependent on a Load

Fig. 10. Detecting If Memory Addresses Are the Same, Differ, or Dependent

(3) There can be no possibility of an intervening store to the same address between the load and the store. Fig. 10(b) shows a load of a variable `v` followed by a store to the same variable with an intervening store. The compiler must ensure that the value in `r[c]` is not the address of the variable `v`. When our analysis could not guarantee a different address in an intervening store, the transformation was not performed.

(4) The value in `r[b]` that will be stored cannot depend on the value loaded into `r[a]`, as shown in Fig. 10(c). This would create a dependency between the load and the store and would prevent the two operations from being performed in parallel. Given available registers, we were able to eliminate false dependences between the load and the store via register renaming. Consider the unrolled loop in Fig. 4(b). Fig. 11(a) shows the same loop, but in a load/store fashion, where the temporaries are registers. The load and store cannot be made contiguous due to reuse of the same registers. Fig. 11(b) shows the same code after the compiler renamed the registers on which the value to be stored depends. Now the instructions can be scheduled so that the load and store can be made contiguous as shown in Fig. 11(c). Fig. 11(d) shows the load and store coalesced and the loaded register renamed.

(5) 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. 12(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. 12(b) shows that our compiler is able to reschedule the instructions between the load and the store to meet this condition. Now the load and the store can be moved where the load appears immediately before the store, as shown in Fig. 12(c). Once the load and store are contiguous, the two instructions can be coalesced. Fig. 12(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]`.

(6) The value in the register to be stored cannot be used after the store instruction. Consider the example in Figure 13(a), where `r[b]` is used after the store. Figure 13(b) shows that coalescing the load and store can result in a different value in `r[b]`.

```
for (j = n-1; j > 1; j -= 2) {
  r[1] = d[j-1];
  r[2] = dd[j];
  r[1] = r[1]-r[2];
  d[j] = r[1];
  r[1] = d[j-2];
  r[2] = dd[j-1];
  r[1] = r[1]-r[2];
  d[j-1] = r[1];
}
```

(a) After Loop Unrolling

```
for (j = n-1; j > 1; j -= 2) {
   r[1] = d[j-1];
   r[2] = dd[j];
   r[1] = r[1]-r[2];
   d[j] = r[1];
   r[3] = dd[j-1];
   r[4] = d[j-2];
   r[3] = r[3]-r[4];
   d[j-1] = r[3] ;
}
```

(b) After Register Renaming

```
for (j = n-1; j > 1; j -= 2) {
   r[3] = d[j-2];
   r[4] = dd[j-1];
   r[3] = r[3]-r[4];
   r[1] = d[j-1];
   d[j-1] = r[3];
   r[2] = dd[j];
   r[1] = r[1]-r[2];
   d[j] = r[1];
}
```

(c) After Scheduling the Instructions

```
for (j = n-1; j > 1; j -= 2) {
   r[3] = d[j-2];
   r[4] = dd[j-1];
   r[3] = r[3]-r[4];
   r[3] = d[j-1]; d[j-1] = r[3];
   r[2] = dd[j];
   r[1] = r[3]-r[2];
   d[j] = r[1];
}
```

(d) After Coalescing the Load and Store

Fig. 11. Applying Register Renaming to Permit the Use of a Swap Instruction

```
r[a] = M[v];
...
... = ... r[a] ...;
... = ... r[b] ...;
...
M[v] = r[b];
```

(a) Use of r[a] Appears before a Reference to r[b]

```
r[a] = M[v];
...
... = ... r[b] ...;
... = ... r[a] ...;
...
M[v] = r[b];
```

(b) First Use of r[a] Appears after the Last Reference to r[b]

```
...
... = ... r[b] ...;
r[a] = M[v];
M[v] = r[b];
... = ... r[a] ...;
...
```

(c) Load and Store Can Now Be Made Contiguous

```
...
... = ... r[b] ...;
r[b] = M[v]; M[v] = r[b];
... = ... r[b] ...;
...
```

(d) After Coalescing the Load and Store and Renaming r[a] to r[b]

Fig. 12. Scheduling Instructions to Exploit a Swap

```
r[a] = M[v];
...
M[v] = r[b];
...
r[c] = r[b];
```

(a) Use of r[b] Following Store

```
...
r[b] = M[v]; M[v] = r[b];
...
r[c] = r[b];
```

(b) Illegal Coalescing

Fig. 13. The Value to be Stored Must Not Be Used after the Store

```
r[a] = M[v];
...
M[v] = r[b];
...
r[b] = ...;
...
... = ... r[a] ...;
...
... = ... r[b] ...;
```

(a) r[b] Is Set in the Live Range of r[a]

```
r[a] = M[v];
...
M[v] = r[b];
...
r[c] = ...;
...
... = ... r[a] ...;
...
... = ... r[c] ...;
```

(b) Live Range of r[b] after Store Has Been Renamed to r[c]

```
...
r[b] = M[v]; M[v] = r[b];
...
r[c] = ...;
...
... = ... r[b] ...;
...
... = ... r[c] ...;
```

(c) After Coalescing the Load and Store and Renaming the Live Range of r[a] to r[b]

Fig. 14. Another Example of Applying Register Renaming

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

Table 1. Test Programs

(7) The register that was loaded has to be able to be renamed to the register that was stored. Consider the example in Fig. 14(a), which contains a set of `r[b]` after the store and before the last use of the value assigned to `r[a]`. In this situation, we attempt to rename the second live range of `r[b]` to a different available register. Fig. 14(b) shows this live range being renamed to `r[c]`. Fig. 14(c) depicts that the load and store can now be coalesced since `r[a]` can be renamed to `r[b]`.

5 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 [9]. 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.

Program	Instructions Executed			Memory References Performed		
	Not Coalesced	Coalesced	Decrease	Not Coalesced	Coalesced	Decrease
bandec	69,189	68,459	1.06%	18,054	17,324	4.04%
bubblesort	2,439,005	2,376,705	2.55%	498,734	436,434	12.49%
chebpc	7,531,984	7,029,990	6.66%	3,008,052	2,507,056	16.66%
elmhes	18,527	18,044	2.61%	3,010	2,891	3.95%
fft	4,176,112	4,148,112	0.67%	672,132	660,932	1.67%
gaussj	27,143	26,756	1.43%	7,884	7,587	3.77%
indexx	70,322	68,676	2.34%	17,132	15,981	6.72%
ludcmp	10,521,952	10,439,152	0.79%	854,915	845,715	1.08%
mmid	267,563	258,554	3.37%	88,622	79,613	10.17%
predic	40,827	38,927	4.65%	13,894	11,994	13.67%
rtflsp	81,117	80,116	1.23%	66,184	65,183	1.51%
select	19,939	19,434	2.53%	3,618	3,121	13.74%
thresh	7,958,909	7,661,796	3.73%	1,523,554	1,226,594	19.49%
transpose	42,883	37,933	11.54%	19,832	14,882	24.96%
traverse	94,159	91,090	3.26%	98,311	96,265	2.08%
tsp	64,294,814	63,950,122	0.54%	52,144,375	51,969,529	0.34%
average	6,103,402	6,019,616	3.06%	3,689,893	3,622,568	8.52%

Table 2. Results

Measurements were collected using the *ease* system that is available with the *vpo* compiler. In some cases, we emulated a swap instruction when it did not exist. For instance, the SPARC does not have swap instructions that swaps bytes, halfwords, floats, or doublewords. The *ease* system provides the ability to gather measurements on proposed architectural features that do not exist on a host machine [2, 3]. Note that it is sometimes possible to use the SPARC swap instruction, which exchanges a word in an integer register with a word in memory, for exchanging a floating-point value with a value in memory. When the floating-point values that are loaded and stored are not used in any operations, then these values could be loaded and stored using integer registers instead of floating-point registers and the swap instruction could be exploited.

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. However, the data cache work decreased by 7.81%, where each hit required one cycle and each miss required 10 cycles. Due to emulating some of the swap instructions, execution time measurements could not be obtained. The actual effect on the execution time would depend on the implementation of the swap instruction and the relative time required for the execution of a swap versus the time required for a load and a store.

6 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 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. We have discussed how a swap instruction could be efficiently integrated into a conventional load/store architecture. A number of different types of opportunities for exploiting the swap instruction were shown to be available. The conditions for coalescing a load and a store into a swap instruction were given and 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.

References

1. M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN'88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, pp. 329–338, June 1988.
2. J.W. Davidson and D.B. Whalley, "Ease: An Environment for Architecture Study and Experimentation," *Proceedings SIGMETRICS'90 Conference on Measurement and Modeling of Computer Systems,* Pages 259–260, May 1990.
3. J.W. Davidson and D.B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions," *Microprocessors and Microsystems*, 15(9), pp. 459–472, November 1991.
4. J.W. Davidson and S. Jinturkar, "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses," *Proceedings of the SIGPLAN'94 Symposium on Programming Language Design and Implementation,* pp. 186–195, June 1994.
5. B. Dwyer, "Simple Algorithms for Traversing a Tree without a Stack," *Information Processing Letters*, 2(5), pp. 143–145, 1973.
6. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann, San Francisco, CA, 1996.
7. M. D. Hill, "A Case for Direct–Mapped Caches," *IEEE Computer*, 21(11), pp. 25–40, December 1988.
8. I. Pitas, *Digital Image Processing Algorithms and Applications,* John Wiley & Sons, Inc., New York, NY, 2000.
9. W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*, Cambridge University Press, New York, NY, 1996.
10. Texas Instruments, Inc., *Product Preview of the TMS390S10 Integrated SPARC Processor*, 1993.