Inter-Iteration Scalar Replacement in the Presence of Conditional Control-Flow

Mihai Budiu
mbudiu@microsoft.com
Microsoft Research, Silicon Valley

Seth Copen Goldstein
seth@cs.cmu.edu
Carnegie Mellon University

Abstract

A large class of multimedia programs for embedded systems manipulate data represented as dense matrices. In this paper we revisit the classical optimization of scalar replacement of array elements and pointer accesses; this optimization allocates array elements to registers, reducing memory traffic. We generalize the state-of-the-art algorithm, by Carr and Kennedy [CK94], improving it to handle simultaneously both conditional control-flow and inter-iteration data reuse. Our algorithm operates within the same assumptions of the classical one (perfect dependence information), and has the same limitations (increased register pressure). It is, however, optimal in the sense that within each code region where scalar promotion is applied, given sufficient registers, each memory location is read/written at most once.

1 Introduction

The goal of scalar replacement (also called register promotion) is to identify repeated accesses made to the same memory address, either within an iteration or across iterations, and to remove the redundant accesses by keeping the data in registers. This optimization was devised in the contex of FORTRAN programs manipulating dense matrices, which exhibit regular array accesses. Many C embedded programs manipulating data and media streams exhibit similar access patterns. Scalar replacement allocates array elements in registers, replacing repeated memory accesses with register file accesses. The algorithm we propose can take advantage of hardware support such as predication, conditional moves, and rotating register files, but can also be implemented purely in software.

We focus in this paper on promotion within the innermost loop bodies, but the ideas we present are applicable to wider code regions as well. The state-of-the-art algorithm for scalar replacement was proposed in 1994 by Steve Carr and Ken Kennedy [CK94][1]. This algorithm handles two special instances of the scalar replacement problem very well: (1) repeated accesses made within the same loop iteration in code having arbitrary conditional control-flow, and (2) code with repeated accesses made across iterations *in the absence of conditional control-flow*. For (1) the algorithm relies on partial redundancy elimination (PRE), while for (2) it relies on dependence analysis and rotating scalar values. That algorithm however cannot handle optimally all cases involving a combination of both conditional control-flow and inter-iteration reuse of data.

We propose a simple algorithm which generalizes and simplifies the Carr-Kennedy algorithm in an optimal way. The optimality criterion is the number of dynamically executed memory accesses; after application of our algorithm on a code region no memory location is read/written more than once in that region. Also, after promotion, no memory location is read or written if it was not in the original program, i.e., our algorithm does not perform speculative promotion. Our algorithm operates under the same assumptions as the Carr-Kennedy algorithm, that is, it requires perfect dependence information to be applicable.

The key idea of the algorithm is to let the compiler create for each value to be scalarized a 1-bit runtime flag variable indicating whether the scalar value is "valid". The compiler also creates code that dynamically updates the flag; the flag is then used to detect and avoid redundant loads and to indicate whether a store has to occur to update a modified value at loop completion. This algorithm ensures that only the first load of a memory location is executed and only the last store takes place. This algorithm relies on a generalization of the technique of predicated partial redundancy elimination (PPRE) proposed by Scholz et al. [SMH03b]: in the same way that the Carr-Kennedy algorithm generalizes PRE to reuse data across remote iterations, our algorithm generalizes PPRE to work even in the presence of unanalyzable control-flow.

[1] In this paper we do not consider speculative promotion, which has been extensively studied since then.

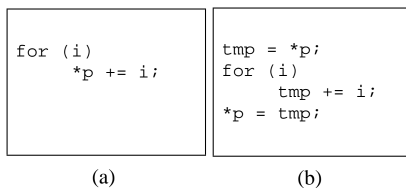


Figure 1: *Sample program with loop invariant memory accesses and its optimization by register promotion.*

We describe the algorithm by a series of examples. In Section 2 we show how the algorithm handles memory operations from loop-invariant addresses. In Section 3.3 we show how to also optimize loads and stores whose addresses are induction variables. In Section 8, we quantify the impact of an implementation of this algorithm when applied to the innermost loops of a series of C programs.

This paper makes several contributions: (1) it introduces a new register-promotion algorithm based on PPRE; (2) it introduces a linear-time term-rewriting algorithm for inter-iteration register promotion in the presence of control-flow; (3) it describes (in Section 5) a novel algorithm based on predicate manipulation to identify and remove loop-invariant accesses; and (4) it evaluates the implementation of this algorithm in an experimental compiler.

Conventions: We present all the optimization examples as source-to-source transformations of schematic C program fragments. For simplicity of the exposition we assume that we are optimizing the body of an innermost loop. We also assume that none of the scalar variables in our examples have their address taken. We write `f(i)` to denote an arbitrary expression involving `i`, which has no side effects. We write `for(i)` to denote a loop having `i` as a basic induction variable. For pedagogical purposes, the examples we present all assume that the code has been brought into a canonical form through the use of *if-conversion* [AKPW83]. However, the algorithm from Section 4 operates on code with arbitrary control-flow. The implementation we evaluate in Section 8 uses if-conversion and predication.

2 Loop-invariant Addresses

2.1 No Control-flow

Figure 1 shows a simple example (a) and how it is transformed (b) by the classical scalar promotion algorithm. Assuming `p` cannot point to `i`, the key fact is `*p` always loads from and stores to the same address, therefore `*p` can be transformed into a scalar value. The load is lifted to the loop pre-header, while the store is moved after the loop. (The latter is slightly more difficult to accomplish if the loop has multiple exits going to multiple destinations. Our implementation handles these as well.)

2.2 Loads and Control-flow

However, the simple algorithm is no longer applicable to the slightly different example in Figure 2(a). Lifting the load or store out of the loop may be unsafe with respect to exceptions: one cannot lift a memory operation out of a loop it if may never be executed within the loop. To optimize this case we can employ the technique of PPRE, by maintaining a `valid` bit in addition to the `tmp` scalar, as shown in Figure 2(b). The `valid` bit indicates whether `tmp` indeed holds the value of `*p`. The `valid` bit is initialized to *false*. A load from `*p` is performed only if the `valid` bit is *false*. Either loading from or storing to `*p` sets the `valid` bit to *true*.

The `valid` flag within an iteration is the dynamic equivalent of the *availability* dataflow information for the loaded value, which is the basis of classical PRE. When PRE can be applied statically, it is certainly better to do so. The problem with Figure 2 is that the compiler cannot statically summarize when condition `(i&1)` is true, and therefore has to act conservatively, assuming that the loaded value is never available. Computing the availability information at run-time eliminates this conservative approximation. Maintaining and using runtime dataflow information makes sense when we can eliminate costly operations (e.g., memory accesses) by using inexpensive operations (e.g., Boolean register operations).

2.3 Loop-invariant Addresses for Stores

This algorithm generates a program which is optimal with respect to the number of loads within each region of code to which promotion is applied (if the original program loads from an address, then the optimized program will load from that address exactly once), but may execute one extra store:[2] if the original program loads the value but never stores to it, the `valid` bit will be true, enabling the postlude store. To treat this case as well, a `dirty` flag, set on writes, has to be maintained, as shown in Figure 2(c).[3] This program will forward the value of `*p` through the scalar `tmp` between iterations arbitrarily far apart. Note that PPRE alone is unable to optimize redundant stores.

[2] However, this particular program is optimal for stores as well.

[3] To simplify the presentation, the examples in the rest of the paper will not include the `dirty` bit. However, its presence is required for achieving an optimal number of stores. The `dirty` bit may also be required for correctness, if the value is read-only and the writes within the loop are always dynamically predicated "false."

```
for (i)
    if (i & 1)
        *p += i;
```

(a) Hard example.

```
/* prelude */
tmp_valid = false;
for (i) {
    /* load from *p */
    if ((i & 1) && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }
    /* store to *p */
    if (i & 1) {
        tmp += i;
        tmp_valid = true;
    }
}
/* postlude */
if (tmp_valid)
    *p = tmp;
```

(b) Loop-invariant loads.

```
/* prelude */
tmp_valid = false;
tmp_dirty = false;

for (i) {
    /* load from *p becomes: */
    if ((i & 1) && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* store to *p becomes */
    if (i & 1) {
        tmp += i;
        tmp_valid = true;
        tmp_dirty = true;
    }
}

/* postlude */
if (tmp_dirty)
    *p = tmp;
```

(c) Loop-invariant stores.

Figure 2: *A small program that is not amenable to classical register promotion and its optimization.*

```
for (i = 2; i < N; i++)
    a[i] = a[i] + a[i-2];
```

(a)

```
/* pre-header */
a0 = a[0];      /* invariant  a0 = a[i-2] */
a1 = a[1];      /*            a1 = a[i-1] */
for (i = 2; i < N; i++) {
    a2 = a[i];  /*            a2 = a[ i ] */
    a2 = a0 + a2;
    a[i] = a2;

    /* Rotate scalar values */
    a0 = a1;
    a1 = a2;
}
```

(b)

Figure 3: *Sample program before and after optimization by register promotion using the Carr-Kennedy algorithm.*

3 Scalar promotion

3.1 The Carr-Kennedy Algorithm

Figure 3(b) illustrates the result of applying the classical Carr-Kennedy [CCK90] inter-iteration register promotion algorithm to Figure 3(a). In general, reusing a value after k iterations requires the creation of k distinct scalar values, to hold the simultaneously live values of `a[i]` loaded for k consecutive values of `i`. This quickly creates register pressure, and thus heuristics are usually used to decide whether promotion is beneficial. Since register pressure has been very well addressed in the literature [CCK90, Muc97, CMS96, CW95], we will not further discuss it in this text.

An extension to the Carr-Kennedy algorithm [CK94] allows it to handle control flow; by using PRE on the loop body it achieves optimality for values reused *within* the same iteration. However, in general it can not promote values *across* iterations in the presence of control-flow. The compiler has difficulty in reasoning about the intervening updates between accesses made in different iterations in the presence of control-flow (more precisely, it won't be able to promote values if dependence distances are not "consistent").

3.2 Partial Redundancy Elimination (PRE)

Before presenting our solution let us note that even the classical PRE algorithm (without the support of special register promotion) is quite successful in optimizing loads made in *consecutive* iterations. The `gcc` compiler, which does *not* have a register promotion algorithm, optimizes Figure 4(a) as in Figure 4(b). Using PRE `gcc` manages to reuse the load from `ptr2` one iteration later.

The PRE algorithm (and its generalization, PPRE) is unable to achieve the same effect if data is reused in any iteration other than the immediately following iteration or if there are intervening stores. In such cases an algorithm like Carr-Kennedy is necessary to remove the redundant accesses. Notice that the use of `valid` flags achieves the

```
do {
    *ptr1++ = *ptr2++;
} while(--cnt && *ptr2);
```

(a) before

```
tmp = *ptr2;
do {
    *ptr1++ = tmp;
    ptr2++;
    if (--cnt) break;
    tmp = *ptr2;
    if (! tmp) break;
} while(1);
```

(b) after

Figure 4: *Sample program optimized by gcc using PRE.*

same degree of optimality as PRE *within* an iteration, but at the expense of maintaining run-time information.

3.3 Removing All Redundant Loads

The classical algorithm is unable to promote all memory references guarded by a conditional, as in Figure 5(a). It is, in general, impossible for a compiler to check when `f(i)` is true in both iteration `i` and in iteration `i-2`, and therefore it cannot deduce whether the load from `a[i]` can be reused as `a[i-2]` two iterations later.

Register promotion has the goal of only executing the *first* load and the *last* store of a variable. Our algorithm for handling loop-invariant data is immediately applicable for promoting loads across iterations, since it performs a load as soon as possible. By maintaining availability information at runtime, using `valid` flags, our algorithm can transform the code to perform a minimal number of loads as in Figure 5(b). (Applying constant propagation and dead-code elimination will simplify this code by further removing the unnecessary references to `a2_valid`.)

3.4 Removing All Redundant Stores

Stores should not be performed if their value will be overwritten in a subsequent iteration. In the presence of control-flow it is not obvious how to deduce whether the overwriting stores in future iterations will take place. For the example in Figure 6(b), (which is the result of applying the algorithm as described so far to Figure 6(a)), we want to avoid storing to `a[i+2]`, since that store will be overwritten two iterations later by the store to `a[i]`. However, this is not true for the last two iterations of the loop. Since, in general, the compiler cannot generate code to test loop-termination several iterations ahead,

```
for (i = 2; i < N; i++)
  if (f(i))
    a[i] = a[i]+a[i-2];
```

(a) Strided memory accesses.

```
a0_valid = false;
a1_valid = false;
a2_valid = false;
for (i) {
  fi = f(i);

  /* load a[i-2] */
  if (fi && !a0_valid) {
    a0 = a[i-2];
    a0_valid = true;
  }

  /* load a[i] */
  if (fi && !a2_valid) {
    a2 = a[i];
    a2_valid = true;
  }

  /* store a[i] */
  if (fi) {
    a2 = a0 + a2;
    a[i] = a2;
    a2_valid = true;
  }

  a0 = a1;   a1 = a2;
  a0_valid = a1_valid;
  a1_valid = a2_valid;
  a2_valid = false;
}
```

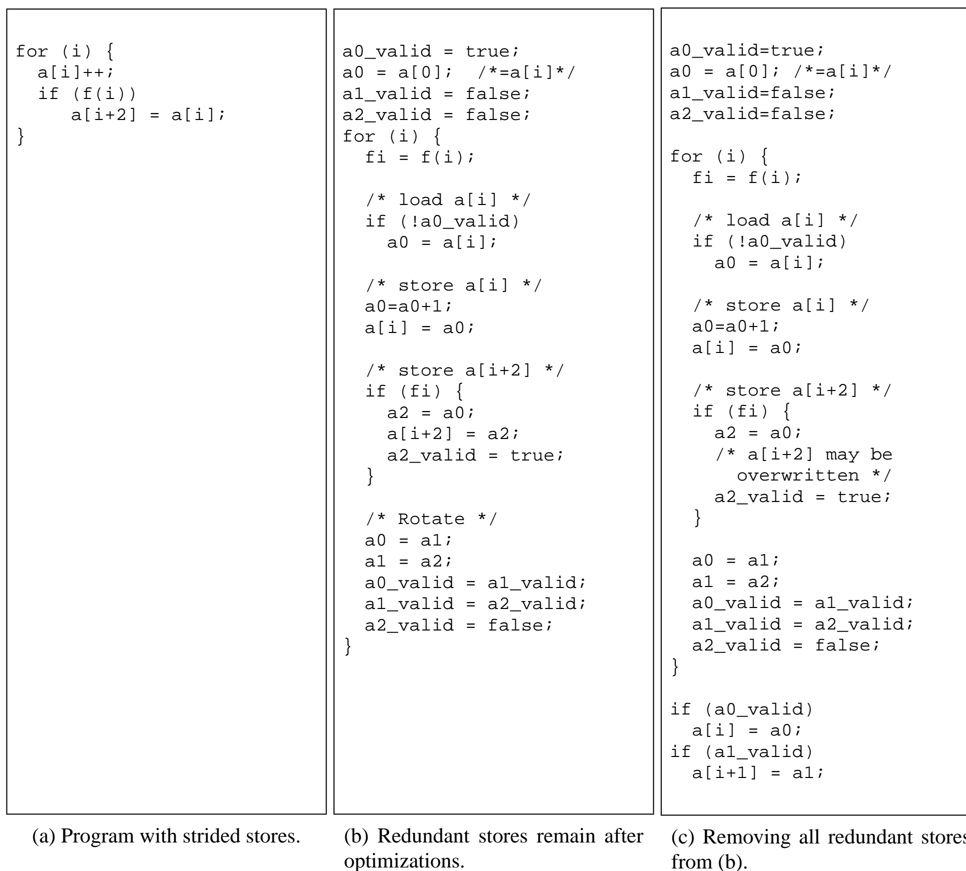
(b) Optimizing strided loads.

Figure 5: *Optimizing strided loads.*

it looks as if both stores must be performed in each iteration. However, we can do better than that by performing, within the loop, only the store to `a[i]`, which certainly will not be overwritten. The loop in Figure 6(c) does exactly that. The loop body never overwrites a stored value but may fail to correctly update the last two elements of array `a`. Fortuitously, after the loop completes, the scalars `a0`, `a1` hold exactly these two values. So we can insert a loop postlude to fix the potentially missing writes. (Of course, `dirty` bits should be used to prevent useless updates.)

4 The register promotion algorithm

The hard work consists in computing the data dependences and deciding whether promotion is applicable; any traditional method can be applied for this purpose. We only describe the code transformations for achieving promotion. In general, for each reference to `a[i+`j`]` (for a compile-time constant j) we maintain a scalar $\mathtt{t}_j$ and



```
for (i) {
  a[i]++;
  if (f(i))
     a[i+2] = a[i];
}
```

(a) Program with strided stores.

```
a0_valid = true;
a0 = a[0];  /*=a[i]*/
a1_valid = false;
a2_valid = false;
for (i) {
  fi = f(i);

  /* load a[i] */
  if (!a0_valid)
    a0 = a[i];

  /* store a[i] */
  a0=a0+1;
  a[i] = a0;

  /* store a[i+2] */
  if (fi) {
    a2 = a0;
    a[i+2] = a2;
    a2_valid = true;
  }

  /* Rotate */
  a0 = a1;
  a1 = a2;
  a0_valid = a1_valid;
  a1_valid = a2_valid;
  a2_valid = false;
}
```

(b) Redundant stores remain after optimizations.

```
a0_valid=true;
a0 = a[0]; /*=a[i]*/
a1_valid=false;
a2_valid=false;

for (i) {
  fi = f(i);

  /* load a[i] */
  if (!a0_valid)
    a0 = a[i];

  /* store a[i] */
  a0=a0+1;
  a[i] = a0;

  /* store a[i+2] */
  if (fi) {
    a2 = a0;
    /* a[i+2] may be
       overwritten */
    a2_valid = true;
  }

  a0 = a1;
  a1 = a2;
  a0_valid = a1_valid;
  a1_valid = a2_valid;
  a2_valid = false;
}

if (a0_valid)
  a[i] = a0;
if (a1_valid)
  a[i+1] = a1;
```

(c) Removing all redundant stores from (b).

Figure 6: *Optimizing strided stores.*

a valid bit $\mathtt{t}_j\mathtt{valid}$. The bits are all initialized to 0 in the loop prelude. Then scalar replacement just makes the following replacements:

Ld `a[i+`j`]` $\rightarrow$ $\mathtt{t}_j$`=`$\mathtt{t}_j$`valid?`$\mathtt{t}_j$`:a[i+`j`];`$\mathtt{t}_j$`valid=1`
St `a[i+`j`]=e` $\rightarrow$ $\mathtt{t}_j$ `= e;` $\mathtt{t}_j$`valid=1`

Furthermore, all stores except the generating store[4] are removed; instead compensation code is added "after" the loop appending for each $\mathtt{t}_j$ a statement "`if (`$\mathtt{t}_j$`valid) a[i+`j`] = `$\mathtt{t}_j$". Code to rotate the scalars and flags is added at the end of each iteration. It is very important to notice that the presence of predication or conditional moves enables the above code transformations to be made *without introducing additional branches*.

Complexity: the algorithm, aside from the dependence analysis, is linear in the size of the loop (we assume that only a constant number of scalar values is created).

Correctness and optimality follow from the following ***invariant:*** the $\mathtt{t}_j\mathtt{valid}$ flag is dynamically "true" iff $\mathtt{t}_j$ represents the contents of the memory location it scalarizes.

5 Handling loop-invariant predicates

The register promotion algorithm described above can be improved by recognizing loop-invariant predicates[5]. If the disjunction of the predicates guarding all the loads and stores with the same address contains a loop-invariant subexpression, then the initialization load can be lifted out of the loop guarded by that subexpression. Figure 7(a) illustrates such a case. Here `c1` and `c2` stand for loop-invariant expressions. The basic algorithm described above, which doesn't optimize loop-invariant predicates specially, generates the code in Figure 7(b). By using the fact that `c1` and `c2` are loop-invariant the code can be optimized as in Figure 7(c). This optimization is mostly effective when applied to loop-invariant memory accesses.

[4] According to the terminology in [CCK90], a generating store is the one writing to `a[i+`j`]` for the smallest j promoted.

[5] Recall that we assume that the code has been if-converted.

```
for (i) {
    if (c1) *p += 1;
    if (c2) *p += 2;
    if (f(i)) *p += i;
}
```

(a) Program with loop-invariant predicates.

```
tmp_valid = false;
for (i) {
    /* first load from *p */
    if (! tmp_valid && c1) {
        tmp = *p;
        tmp_valid = true;
    }

    /* first store to *p */
    if (c1) {
        tmp += 1;
        tmp_valid = true;
    }

    /* 2nd load from *p */
    if (! tmp_valid && c2) {
        tmp = *p;
        tmp_valid = true;
    }

    /* 2nd store to *p */
    if (c2) {
        tmp += 2;
        tmp_valid = true;
    }

    fi = f(i);

    /* 3rd load from *p */
    if (fi && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* 3rd store to *p */
    if (fi) {
        tmp += i;
        tmp_valid = true;
    }
}
if (tmp_valid)
    *p = tmp;
```

(b) Optimization according to described algorithm.

```
/* prelude */
tmp_valid = c1 || c2;
if (tmp_valid)
    tmp = *p;

for (i) {
    /* 1st load *p redundant */

    /* 1st store to *p */
    if (c1)
        /* tmp_valid is true */
        tmp += 1;

    /* 2nd load *p redundant */

    /* 2nd store to *p */
    if (c2)
        tmp += 2;

    fi = f(i);

    /* 3rd load from *p */
    if (fi && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

    /* 3rd store to *p */
    if (fi) {
        tmp += i;
        tmp_valid = true;
    }
}

/* postlude */
if (tmp_valid)
    *p = tmp;
```

(c) Lifting loop-invariant components.

Figure 7: *Optimizing loop-invariant guarding predicates.*

The programs in Figure 7(b) and (c) both execute the same number of loads and stores, and thus, by our optimality criterion are equally good. However, (c) executes fewer total instructions.

We can also lift operations out of the loop when the disjunction of all conditions guarding loads or stores from `*p` is weaker than some loop-invariant expression, even when none of the conditions is itself loop-invariant. Figure 8(a) shows such an example. The disjunction of all predicates is `(f(i)||!f(i))` which is "true", and thus the load from `*p` can be unconditionally lifted out of the loop, generating the result in Figure 8(b).

The general algorithm for lifting initializations out of the loop is the following: let us assume that each statement s is controlled by predicate $P(s)$. Then for each promoted memory location `a[i+`j`]`:

(1) Define the predicate $P_j = \vee_{s_j} P(s_j)$, where $s_j \in \{$statements accessing `a[i+`j`]`$\}$.
(2) Write P_j as the union of two predicates, $P_j^{inv} \vee P_j^{var}$, where P_j^{inv} is loop-invariant and P_j^{var} is loop-dependent.
(3) In prelude initialize $\texttt{t}_j\texttt{_valid} = P_j^{inv}$.
(4) In prelude initialize $\texttt{t}_j\texttt{=t}_j\texttt{_valid?a[i}_0\texttt{+}j\texttt{]:0}$, where $\texttt{i}_0$ is the initial value of `i` in the loop.
(5) The predicate of each statement $P(s_j)$ is strengthened to $P(s_j) \wedge \neg P_j^{inv}$.

6 Hardware support

While our algorithm does not require any special hardware support, certain hardware structures can improve its efficiency.

```
for (i) {
    if (f(i))
        *p += 1;
    else
        *p = 2;
}
```

(a) Program without loop-invariant predicates.

```
tmp = *p;
for (i) {
    fi = f(i);
    if (fi)
        tmp += 1;
    if (!fi)
        tmp = 2;
}
*p = tmp;
```

(b) Afer complete optimization.

Figure 8: *More optimization of loop-invariant guarding predicates.*

Rotating registers were introduced in the Cydra 5 architecture [DHB89] to support software pipelining. These are used on Itanium for traditional register promotion [DKK+99], to shift all the scalar values in one cycle.

Rotating predicate registers as in the Itanium can rotate the "valid" flags.

Software valid bits can be used to reduce the overhead of maintaining the `valid` bits. If a value is reused k iterations later, then our algorithm requires the use of $2k$ different scalars: k `valid` bits and k values. A software-only solution is to pack the k valid bits into a single integer[6] and to use masking and shifting to manipulate them. This makes rotation very fast, but testing and setting more expensive, a trade-off that may be practical on a wide machine having "free" scheduling slots.

Predicated data [RC03] has been proposed for an embedded VLIW processor: predicates are not attached to instructions, but to data itself, as an extra bit of each register. Predicates are propagated through arithmetic, similar to exception poison bits. The proposed architecture supports rotating registers by implementing the register file as an actual large shift register. These architectural features would make the `valid` flags essentially free both in space and in time.

[6]Most likely promotion across more iterations than bits in an integer requires too many registers to be profitable.

7 Related work

The canonical register promotion papers are by Steve Carr et al.: [CCK90, CK94]. Duesterwald et al. [DGS93] describes a dataflow analysis for analyzing array references; the optimizations based on it are conservative: only busy stores and available loads are removed; they notice that the redundant stores can be removed and compensated by peeling the last k loop iterations, as shown by us in Section 3.4. Lu and Cooper [LC97] study the impact of powerful pointer analysis in C programs for register promotion. Sastry and Lu [SJ98] introduce the idea of selective promotion for analyzable regions. None of these algorithms simultaneously handles both inter-iteration dependences and control-flow in the way suggested in this paper. [SJ98, LCK+98] show how to use SSA to facilitate register promotion. [LCK+98] also shows how PRE can be "dualized" to handle the removal of redundant store operations.

Bodík et al. [BGS99] analyzes the effect of PRE on promoting loaded values and estimates the potential improvements. The idea of predicating code for dynamic optimality was advanced initially by Bodík [BG97], and was applied for partial dead-code elimination. Scholz et al. [SMH03a, SMH03b] use the technique of predication for simplifying partial redundancy elimination, creating PPRE.

In this paper we dualize PPRE enabling it to handle stores as well, and we extend its usage in the style of Carr and Kennedy, to handle multiple iterations. We quantify the effectiveness of the technique by actual measurements on a wide range of C programs.

Schemes that use hardware support for register promotion such as [PGM00, DO94, OG01] are radically different from our proposal, which is software-only.

Muchnick [Muc97] gives an example in which a load can be lifted out of a loop because it occurs on both branches of an `if` statement (which would perform the optimization in Figure 8), but he doesn't describe a general algorithm for solving the problem optimally.

8 Experimental evaluation

8.1 Expected Performance Impact

The scalar promotion algorithm presented here is optimal with respect to the number of loads and stores executed, but this may not necessarily lead to improved performance for four reasons:

(1) the optimized code uses more registers to hold the scalar values and flags, and thus may cause more spill code, or interfere with software pipelining [CW95].

(2) the optimized code contains more computations than the original program, in maintaining the flags. The optimized program may end-up being slower than the original, depending, among other things, on the frequency with which the memory access statements are executed and whether the predicate computations are on the critical path. For example, if none of the memory accesses is executed dynamically, all the code inserted by our algorithm is overhead. In practice, profiling information and heuristics should be used to select the loops which will benefit most from this transformation.

(3) scalar promotion usually removes memory accesses which mostly hit in the cache, thus its benefit is limited. However, in modern architectures not all cache accesses, not even L1 cache hits, are cheap. For example, on the Intel Itanium 2 some L1 cache hits may cost as much as 17 clock cycles [CL03]. Register promotion trades-off bandwidth to the load-store queue for bandwidth to the register file, which is always bigger.

(4) by predicating memory accesses, operations which were originally independent, and could be potentially issued in parallel, become now dependent through the predicates. This could increase the dynamic critical path of the program, especially when memory bandwidth is not a bottleneck.

8.2 Performance Measurements

In this section we present measurements of our register promotion algorithm. The algorithm was completely implemented in the CASH C compiler [BG03], and a detailed description of our implementation, based on a Static-Single Assignment representation [CFR$^+$91], can be found in [BG04]programs from three benchmark suites: Mediabench [LPMS97], SpecInt95 and SpecInt2000. The program compilation is fully automated: no source-code changes are made to the standard input benchmarks. We present data for all the programs which ran on our simulation infrastructure.

Our implementation has the following limitations: it does not use `dirty` bits, and thus is not optimal with respect to the number of stores. Second, it only lifts loop-invariant predicates to guard the initializer, such as in Figure 7, but not as in Figure 8. As a simple heuristic to reduce register pressure, we do not scalarize a value if it is not reused for 3 iterations. The compiler uses a flow-sensitive intra-procedural pointer analysis, which affects the precision of the disambiguation.

Table 1 shows how often scalar promotion can be applied. It separates promotion according to address kind (loop-invariant versus strided) and according to the control-flow constraints (**o**, from "old", indicating whether the code could be handled by the traditional algorithm, and **n**, from "new", indicating the *additional* cases handled only by our improved version). For some benchmarks, such as `go`, `vortex` and `gcc`, the new algorithm is essential for uncovering most of the opportunities.

Bench	**Variables**				**Bench**	**Variables**			
	Inv		**Str**			**Inv**		**Str**	
	o	**n**	**o**	**n**		**o**	**n**	**o**	**n**
adpcm_e	0	0	0	0	go	40	53	2	2
adpcm_d	0	0	0	0	m88ksim	23	10	1	4
gsm_e	1	1	1	0	compress	0	0	1	0
gsm_d	1	1	1	0	li	1	0	1	1
epic_e	0	0	0	0	ijpeg	5	1	9	5
epic_d	0	0	0	0	perl	6	0	0	1
mpeg2_e	1	0	1	0	vortex	22	20	1	0
mpeg2_d	4	3	0	0					
jpeg_e	3	0	7	5	gzip	20	0	1	0
jpeg_d	2	1	7	5	vpr	7	2	0	0
pegwit_e	6	0	3	1	gcc	11	40	5	2
pegwit_d	6	0	3	1	mcf	0	0	0	0
g721_e	0	0	2	0	twolf	1	2	0	0
g721_d	0	0	2	0	parser	20	3	3	5
pgp_e	24	1	5	0	vortex	22	20	1	0
pgp_d	24	1	5	0	bzip2	2	2	8	0
rasta	3	0	2	1	gap	1	0	18	1
mesa	44	4	2	0					

Table 1: *How often scalar promotion is applied.*

Figure 9 shows the decrease in the number of loads and stores respectively resulting from the application of our algorithms; the baseline is the program with no memory optimizations applied. These numbers are independent on the actual target architecture. We are counting *all* memory accesses in the program, and not just the optimized loops. The bottom bar shows the percentage reduction obtained by using only the PRE optimizations from [BG03], and no inter-iteration register promotion. The top bar shows the additional percentage decrease from using the algorithms in this paper. We have included both bars since some of the accesses can be eliminated by both algorithms, and the algorithms in [BG03] must be executed in order to enable register promotion in CASH. For 8 programs register promotion alone removes more than 1% in the number of loads, with a maximum of 36% for gsm_e. There are 5 programs for which register promotion removes more than 1% of the stores, with an impressive maximum of 53.5% for 124.m88ksim. The biggest reductions can be attributed to lifting loop-invariant memory accesses out of loops.

Figure 10 shows the impact of the optimizations on the actual program running time. Unfortunately, CASH's back-end for a traditional CPU is still under development, so we have evaluated this performance using Spatial Computation [BVCG04]. Spatial Computation implements programs directly as hardware circuits. For the purposes of this evaluation, it can be seen as an approximation for a very wide dynamically scheduled processor. We use a traditional, bandwidth-limited memory system, connected through a load-store queue and an L1 cache with a 4-

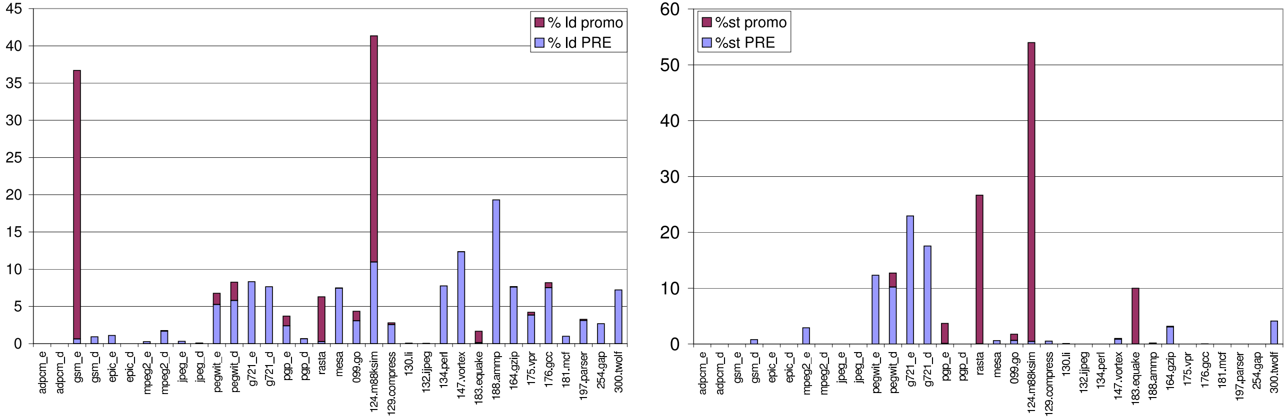


Figure 9: *Percentage reduction in the number of (left) dynamic loads and (right) dynamic stores resulting from the application of our memory optimizations.*

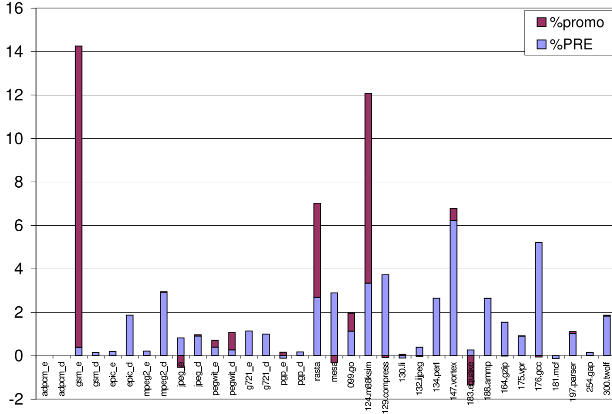


Figure 10: *Percentage reduction in the execution time due to application of our memory optimizations.*

cycles hit time. The other parameters of the memory system are as follows: L1D is a 16K 4-way associative, L2 is a unified cache with 256K, 2-way associative, with a latency of 64 cycles, and the main memory has a latency of 128 cycles, sustaining a throughput of 1 word every four cycles. We use the cache simulator from the SimpleScalar processor simulator [BA97].

The speed-ups range from a 1.1% slowdown for `equake`, to a maximum speed-up of 14% for `gsm_e`. There is a fairly good correlation of speed-up and the number of removed loads. The number of removed stores seems to have very little impact on performance, indicating that the load-store queue contention caused by stores is not a problem for performance (since stores complete asynchronously, they do not have a direct impact on end-to-end performance). 5 programs have a performance improvement of more than 5%. Since most operations removed are relatively inexpensive, because they have good temporal locality, the performance improvement is not very impressive. Register promotion alone causes a slight slow-down for 4 programs, while being responsible for a speed-up of more than 1% for 7 programs.

Interestingly, our algorithm provides better results for a faster memory system (e.g., with a perfect L1 cache with a latency of 2 cycles the `gsm_e` speed-up becomes 18%). This effect occurs because in a slow memory system the improvement obtained by eliminating a load hitting in L1 is smaller than in a system with faster memory.

9 Conclusions

We have described a scalar promotion algorithm which eliminates all redundant loads and stores even in the presence of conditional control flow. The key insight in our algorithm is that availability information, traditionally computed only at compile-time, can be more precisely evaluated at run-time and used to predicate redundant memory accesses. We transform memory accesses into scalar values and perform the loads only when the scalars do not already contain the correct value, and the stores only when their value will not be overwritten. Our approach substantially increases the number of instances when register promotion can be applied.

As the computational bandwidth of processors increases, such optimizations may become more advantageous. In the case of register promotion, the benefit of removing memory operations sometimes outweighs the increase in scalar computations to maintain the dataflow information.

Acknowledgments This research is funded in part by the National Science Foundation under Grants No. CCR-

9876248 and CCR-0205523 by DARPA under contracts MDA972-01-03-0005 and N000140110659 and by the SRC.

References

[AKPW83] R. Allen, Kennedy K., C. Porterfield, and J.D. Warren. Conversion of control dependence to data dependence. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 177–189, Austin, Texas, January 1983.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. In *Computer Architecture News*, volume 25, pages 13–25. ACM SIGARCH, June 1997.

[BG97] Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 159–170, Las Vegas, Nevada, June 1997.

[BG03] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 216–227, San Francisco, CA, March 23–26 2003.

[BG04] Mihai Budiu and Seth Copen Goldstein. Inter-iteration scalar replacement in the presence of conditional control-flow. Technical Report CMU-CS-04-103, Carnegie Mellon University, Department of Computer Science, February 2004.

[BGS99] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Load-reuse analysis: Design and evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, May 1999.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, October 2004.

[CCK90] S. Carr, D. Callahan, and K. Kennedy. Improving register allocation for subscripted variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, White Plains NY, June 1990.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CK94] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software — Practice and Experience*, 24(1), January 1994.

[CL03] Jean-Francois Collard and Daniel Lavery. Optimizations to prevent cache penalties for the Intel Itanium 2 processor. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, March 23-26 2003.

[CMS96] S. Carr, Q. Mangus, and P. Sweany. An experimental evaluation of the sufficiency of scalar replacement algorithms. Technical Report TR96-04, Michigan Technological University, Department of Computer Science, 1996.

[CW95] Steve Carr and Qunyan Wu. The performance of scalar replacement on the HP 715/50. Technical Report TR95-02, Michigan Technological University, Department of Computer Science, 1995.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–77. ACM Press, 1993.

[DHB89] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–38, April 1989.

[DKK+99] Carole Dulong, Rakesh Krishnaiyer, Dattatraya Kulkarni, Daniel Lavery, Wei Li, John Ng, and David Sehr. An overview of the Intel IA-64 compiler. *Intel Technology Journal*, 1999.

[DO94] Peter J. Dahl and Matthew T. O'Keefe. Reducing memory traffic with CRegs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 100–111, November 1994.

[LC97] John Lu and Keith D. Cooper. Register promotion in C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319. ACM Press, 1997.

[LCK+98] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37. ACM Press, 1998.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–335, 1997.

[Muc97] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc, 1997.

[OG01] S. Onder and R. Gupta. Load and store reuse using register file contents. In *ACM International Conference on Supercomputing*, pages 289–302, Sorrento, Naples, Italy, June 2001.

[PGM00] Matthew Postiff, David Greene, and Trevor Mudge. The store-load address table and speculative register promotion. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 235–244. ACM Press, 2000.

[RC03] Davide Rizzo and Osvaldo Colavin. A scalable wide-issue clustered VLIW with a reconfigurable interconnect. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, CA, 2003.

[SJ98] A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–25. ACM Press, 1998.

[SMH03a] Bernhard Scholz, Eduard Mehofer, and Nigel Horspool. Partial redundancy elimination with predication techniques. In *European Conference on Parallel Processing (EUROPAR)*, volume 2790 of *LNCS*, pages 242–250, Klagenfurt, Austria, August 2003. Springer Verlag.

[SMH03b] Bernhard Scholz, Eduard Mehofer, and Nigel Horspool. Predicated partial redundancy elimination using a cost analysis. *Parallel Processing Letters*, 13(4):525–536, 2003.