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

Mihai Budiu Seth Copen Goldstein

February 1, 2004

CMU-CS-04-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We revisit the classical problem of scalar replacement of array elements and pointer accesses. We generalize the state-of-the-art algorithm, by Carr and Kennedy [CK94], to handle a combination of both conditional control-flow and inter-iteration data reuse. The basis of our algorithm is to make the dataflow availability information precise using a technique we call SIDE: Statically Instantiate and Dynamically Evaluate. In SIDE the compiler inserts explicit code to evaluate the dataflow information at runtime.

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 and written at most once.

This research was sponsored by the Defense Advanced Projects Agency (DARPA), the US Army under contract no. BT6396C0083, the US Navy under grant no. N000140110659, Hewlett Packard, and the National Science Foundation (NSF) under grant no. CCR-0205523. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsors or any other entity.

Keywords: optimizing compilers, register promotion, predication, dataflow analysis

1 Introduction

The goal of scalar replacement (also called register promotion) is to identify repeated accesses made to the same memory address, either within the same iteration or across iterations, and to remove the redundant accesses (here we only study 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 very well two special instances of the scalar replacement problem: (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 PRE, while for (2) it relies on dependence analysis and rotating scalar values. However, that algorithm cannot handle arbitrary combinations of both conditional control-flow and inter-iteration reuse of data.

Here we present a very simple algorithm which generalizes and simplifies the Carr-Kennedy algorithm in an optimal way. The optimality criterion that we use throughout this paper is the number of dynamically executed memory accesses. After application of our algorithm on a code region, no memory location is read more than once and written more than once in that region. Also, after promotion, no memory location is read or written if it was not so 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. It is therefore mostly suitable for FORTRAN benchmarks. We have implemented our algorithm in a C compiler, and we have found numerous instances where it is applicable as well.

For the impatient reader, the key idea is the following: for each value to be scalarized, the compiler creates a 1-bit runtime flag variable indicating whether the scalar value is "valid." The compiler also creates code which 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 is a particular instance of a new general class of algorithms: it transforms values customarily used only at compile-time for dataflow analysis into dynamic objects. Our algorithm instantiates *availability* dataflow information into run-time objects, therefore achieving dynamic optimality even in the presence of constructs which cannot be statically optimized.

We introduce the algorithm by a series of examples which show how it is applied to increasingly complicated code structures. We start in Section 2 by showing how the algorithm handles a special case, that of memory operations from loop-invariant addresses. In Section 3.3 we show how the algorithm optimizes loads whose addresses are induction variables. Finally, we show how stores can be treated optimally in Section 3.4. In Section 4 we describe two implementations of our algorithm: one based on control-flow graphs (CGFs), and one relying on a special form of Static-Single Assignment(SSA) named Pegasus. Although the CFG variant is simpler to implement, Pegasus simplifies the dependence analysis required to determine whether promotion is applicable. Special handling of loop-invariant guarding predicates is discussed in Section 5. Finally, in Section 7, 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 the following new research contributions:

- it introduces the SIDE class of dataflow analyses, in which the analysis is carried statically, but the computation of the dataflow information is performed dynamically, creating dynamically optimal code for constructs which cannot be statically made optimal;

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

```
while (1) {
    if (cond1) statement1;
    if (cond2) statement2;
    ...
    if (cond_break1) break;
    ...
}
```

Figure 1: *For ease of presentation we assume that prior to register promotion, all loop bodies are predicated.*

- it introduces a new register-promotion algorithm as a SIDE dataflow analysis;
- it introduces a linear-time[2] term-rewriting algorithm for performing inter-iteration register promotion in the presence of control-flow;
- it describes register promotion as implemented in Pegasus, showing how it takes advantage of the memory dependence representation for effective dependence analysis.

1.1 Conventions

We present all the optimizations 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 (but *not* a function call). We write `for(i)` to denote a loop having `i` as a basic induction variable; we assume that the loop body is executed at least once.

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], such that each memory statement is guarded by a predicate; i.e., the code has the shape in Figure 1. Our algorithms are easily generalized to handle nested natural loops and arbitrary forward control-flow within the loop body.

2 Scalar Replacement of Loop-Invariant Memory Operations

In this section we describe a new register promotion algorithm which can eliminate memory references made to loop-invariant addresses in the presence of control flow. This algorithm is further expanded in Section 3.3 and Section 3.4 to promote memory accesses into scalars when the memory references have a constant stride.

2.1 The Classical Algorithm

Figure 2 shows a simple example and how it is transformed 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

[2]This time does not include the time to compute the dependences, only the actual register promotion transformation.

```
for (i)
    *p += i;

-------------

tmp = *p;
for (i)
    tmp += i;
*p = tmp;
```

Figure 2: *A simple program before and after register promotion of loop-invariant memory operations.*

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

Figure 3: *A small program that is not amenable to classical register promotion.*

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, as described in Section 4.2.2.)

2.2 Loop-Invariant Addresses and Control-Flow

However, the simple algorithm is no longer applicable to the slightly different Figure 3. 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 Figure 3, it is enough to maintain a `valid` bit in addition to the the `tmp` scalar. The `valid` bit indicates whether `tmp` indeed holds the value of `*p`, as in Figure 4. 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*. This program will forward the value of `*p` through the scalar `tmp` between iterations arbitrarily far apart.

The insight is that it may be profitable to compute dataflow information at runtime. For example, the `valid` flag within an iteration is nothing more than the dynamic equivalent of the *availability* dataflow information for the loaded value, which is the basis of classical Partial Redundancy Elimination (PRE) [MR79]. When PRE can be applied statically, it is certainly better to do so. The problem with Figure 3 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).

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

```
/* prelude */
tmp = uninitialized;
tmp_valid = 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;
     }
}

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

Figure 4: *Optimization of the program in Figure 3.*

optimized program will load from that address exactly once), but may execute one extra store:[3] if the original program loads the value but never stores to it, the `valid` bit will be true, enabling the postlude store. In order to treat this case as well, a `dirty` flag, set on writes, has to be maintained, as shown in Figure 5.[4]

Note: in order 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.

3 Inter-Iteration Scalar Promotion

Here we extend the algorithm for promoting loop-invariant operations to perform scalar promotion of pointer and array variables with constant stride. We assume that the code has been subjected to standard dependence analysis prior to scalar promotion.

3.1 The Carr-Kennedy Algorithm

Figure 6 illustrates the classical Carr-Kennedy inter-iteration register promotion algorithm from [CCK90], which is only applicable in the absence of control-flow. 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

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

[4]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."

```
/* prelude */
tmp = uninitialized;
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;
```

Figure 5: *Optimization of store handling from Figure 4.*

consecutive values of `i`. This quickly creates register pressure and therefore 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 concern ourselves with it anymore in this text.

A later extension to the Carr-Kennedy algorithm [CK94] allows it to also handle control flow. The algorithm optimally handles reuse of values *within* the same iteration, by using PRE on the loop body. However, this algorithm can no longer 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.

3.2 Partial Redundancy Elimination

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. Figure 7 shows a sample loop and its optimization by `gcc`, which does *not* have a register promotion algorithm at all. By using PRE alone `gcc` manages to reuse the load from `ptr2` one iteration later.

The PRE algorithm 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. Let us notice that the use of `valid` flags achieves the same degree of optimality as PRE *within* an iteration, but at the expense of maintaining run-time information.

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

--------------------------------------------

/* 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;
}
```

Figure 6: *Program with no control-flow before and afer register promotion performed by the Carr-Kennedy algorithm.*

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

------------------------

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

Figure 7: *Sample loop and its optimization using PRE. (The output is the equivalent of the assembly code generated by* `gcc`*.) PRE can achieve some degree of register promotion for loads.*

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

Figure 8: *Sample program which cannot be handled optimally by either PRE or the classical Carr-Kennedy algorithm.*

3.3 Removing All Redundant Loads

However, the classical algorithm is unable to promote all memory references guarded by a conditional, as in Figure 8. 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. The algorithm in Section 2 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 9. Applying constant propagation and dead-code elimination will simplify this code by removing the unnecessary references to `a2_valid`.

3.4 Removing All Redundant Stores

Handling stores seems to be more difficult, since one should forgo a store if the value will be overwritten in a subsequent iteration. However, in the presence of control-flow it is not obvious how to deduce whether the overwriting stores in future iterations will take place. Here we extend the register promotion algorithm to ensure that only one store is executed to each memory location, by showing how to optimize the example in Figure 10.

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, 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 11 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 Implementation

This algorithm is probably much easier to illustrate than to describe precisely. Since the important message was hopefully conveyed by the examples, we will just briefly sketch the implementation in a CFG-based framework and describe in somewhat more detail the Pegasus implementation.

```
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;
    }

    /* Rotate scalars and valid flags */
    a0 = a1;
    a1 = a2;
    a0_valid = a1_valid;
    a1_valid = a2_valid;
    a2_valid = false;
}
```

Figure 9: *Optimal version of the code in Figure 8.*

4.1 CFG-Based Implementation

In general, for each constant reference to `a[i+`j`]` (for a compile-time constant j) we maintain a scalar $\texttt{t}_j$ and a valid bit $\texttt{t}_j$`valid`. Then scalar replacement just makes the following changes:

- Replaces every load from `a[i+`j`]` with a pair of statements:
  $\texttt{t}_j$ `=` $\texttt{t}_j$`valid ?` $\texttt{t}_j$ `:  a[i+`j`];` $\texttt{t}_j$`valid = true`
- Replace every store `a[i+`j`] = e` with a pair of statements:
  $\texttt{t}_j$ `= e;` $\texttt{t}_j$`valid = true`.

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

---------------------------------------

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 scalars and valid flags */
    a0 = a1;
    a1 = a2;
    a0_valid = a1_valid;
    a1_valid = a2_valid;
    a2_valid = false;
}
```

Figure 10: *Program with control-flow and redundant stores and its optimization for an optimal number of loads. The store to* `a[i+2]` *may be overwritten two iterations later by the store to* `a[i]`.

```
a0_valid = true;  /* a[i] */
a0 = a[0];
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;
        /* No update of a[i+2]: may be overwritten */
        a2_valid = true;
    }

    /* Rotate scalars and valid flags */
    a0 = a1;
    a1 = a2;
    a0_valid = a1_valid;
    a1_valid = a2_valid;
    a2_valid = false;
}

/* Postlude */

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

Figure 11: *Optimal version of the example in Figure 10.*

Furthermore, all stores except the generating store[5] are removed. Instead compensation code is added "after" the loop: for each $\mathtt{t}_j$ append a statement `if (`$\mathtt{t}_j$`valid) a[i+`j`] = `$\mathtt{t}_j$.

Complexity: the algorithm, aside from the dependence analysis, is linear in the size of the loop[6].

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

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

[6] We assume that a constant number of scalar values are introduced.

Compiler pass	LOC
Depdendence analysis	162
Loop-invariant load/store promotion	100
Register promotion of array elements	205
Induction variable analysis	447
Memory disambiguation	646

Table 1: Size of C++ code implementing analyses and optimizations related to register promotion.

4.2 An SSA-based algorithm

We have implemented the above algorithms in the C Compiler named CASH. CASH relies on a Pegasus [BG02b, BG02a, BG03], a dataflow intermediate representation. In this section we briefly describe the main features of Pegasus and then show how it enables a very efficient implementation of register promotion.

As we argued in [BG03], Pegasus enables extremely compact implementations of many important optimizations; register promotion corroborates this statement. In Table 1 we shows the implementation code size of all the analyses and transformations used by CASH for register promotion.

4.2.1 Pegasus

Pegasus represents the program as a directed graph where nodes are operations and edges indicate value flow. Pegasus leverages techniques used in compilers for predicated execution machines [MLC$^+$92] by collecting multiple basic blocks into one hyperblock; each hyperblock is transformed into straight-line code through the use of the predicated static single-assignment (PSSA) form [CSC$^+$00]. Instead of SSA ϕ nodes, within hyperblocks Pegasus uses explicit *multiplexor* (*mux*) nodes; the mux data inputs are the reaching definitions. The mux predicates correspond to the path predicates in PSSA.

Hyperblocks are stitched together into a dataflow graph representing the entire procedure by creating dataflow edges connecting each hyperblock to its successors. Each variable live at the end of a hyperblock gives rise to an *eta* node [OBM90]. Eta nodes have two inputs—a value and a predicate—and one output. When the predicate evaluates to “true,” the input value is moved to the output; when the predicate evaluates to “false,” the input value and the predicate are simply consumed, generating no output. A hyperblock with multiple predecessors receives control from one of several different points; such join points are represented by *merge* nodes.

Operations with side-effects are parameterized with a predicate input, which indicates whether the operation should take place. If the predicate is false, the operation is not executed. Predicate values are indicated in our figures with dotted lines.

The compiler adds dependence edges between operations whose side-effects may not commute. Such edges only carry an explicit synchronization *token* — not data. Operations with memory side-effects (loads, stores, calls, and, returns) all have a token input. When a side-effect operation depends on multiple other operations (e.g., a write operation following a set of reads), it must collect one token from each of them. For this purpose a *combine* operator is used; a combine has multiple token inputs and a single token output; the output is generated after it receives all its inputs. In figures (e.g., see Figure 12) dashed lines indicate token flow and the combine operator is depicted by a “V”. Token edges explicitly encode data flow through memory. In fact, the token network can be interpreted as an SSA form for the memory values, where the

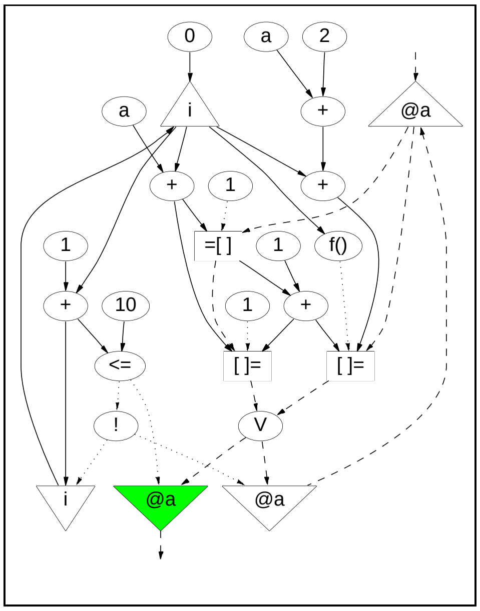

Figure 12: *Pegasus representation of the loop from Figure 8 (the loop bound is 10), before register promotion. Solid lines represent value flow, dotted lines indicate predicate flow and dashed lines represent tokens. The up-triangles are merge operators (i.e., SSA ϕ nodes), and the down-triangles are eta operators. The dark eta sends the token value out of the loop on loop completion. The merge-eta nodes labeled "@a" are used to carry the token mediating all accesses to array* `a`.

combine operator is similar to a ϕ function. The tokens encode both true-, output- and anti-dependences, and they are "may" dependences. In Figure 12(A) there is one load and two stores. A load is denoted by "`=[ ]`" and has 3 inputs: address, predicate and token; it produces two outputs: the loaded value and another token. A store is denoted by "`[ ]=`" and has four inputs: address, data, predicate and token; the only output is a token.

4.2.2 Register Promotion in Pegasus

We sketch the most important analysis and transformation steps carried out by CASH for register promotion. Although the actual promotion in Pegasus is slightly more complicated than in a CFG-based representation (because of the need to maintain ϕ-nodes), the dependence tests used to decide whether promotion can be applied are much simpler: the graph will have a very restricted structure if promotion can be applied.[7] The key element of the representation is the token edge network whose structure can be quickly analyzed to determine important properties of the memory operations.

We illustrate register promotion on the example in Figure 8.

1. The token network for the Pegasus representation is shown in Figure 13. Memory accesses that may interfere with each other will all belong to a same connected component of the token network. Operations that belong to distinct components of the token network commute and can therefore be analyzed separately. In this example there is a single connected component, corresponding to accesses made to the array `a`.

[7]The network encodes relatively simple dependence information. However, as pointed in [CMS96], elementary dependence tests are sufficient for most cases of register promotion.

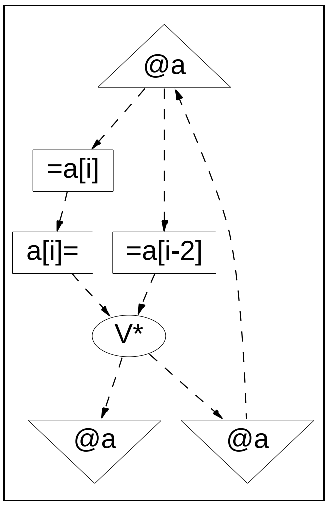

Figure 13: *Token network for the Pegasus representation of the loop in Figure 8.*

2. The addresses of the three memory operations in this component are analyzed: they are all determined to be induction variables having the same step, 1. This implies that the dependence distances between these accesses are constant (i.e., iteration-independent), making these accesses candidates for register promotion.

   The induction step of the addresses indicates the type of promotion: a 0 step indicates loop-invariant accesses, while a non-zero step, as in this example, indicates strided accesses.

3. The token network is further analyzed. Notice that prior to register promotion, memory disambiguation has already proved (based on symbolic computation on address expressions) that the accesses to `a[i]` and `a[i+2]` commute, and therefore there is no token edge between them. The token network for `a` consists of two *strands*: one for the accesses to `a[i]`, and one for `a[i+2]`; the strands are generated at the mu, on top, and joined before the etas, at the bottom, using a combine (V). If and only if all memory accesses within the same strand are made to the same address can promotion be carried.

   CASH generates the initialization for the scalar temporaries and the "valid" bits in the loop pre-header. We do not illustrate this step.

4. Each strand is scanned from top to bottom (from the mu to the eta), term-rewriting each memory operation:

   - Figure 14 shows how a load operation is transformed by register promotion. The resulting construction can be interpreted as follows: "If the data is already valid do not do the load (i.e., the load predicate is 'and'-ed with the negation of the valid bit) and use the data. Otherwise do the load if its predicate indicates it needs to be executed." The multiplexor will select either the load output or the initial data, depending on the predicates. If neither predicate is true, the output of the mux is not defined, and the resulting `valid` bit is false.
   - Figure 15 shows the term-rewriting process for a store. After this transformation, all stores except the generating store are removed from the graph (for this purpose the token input is connected directly to the token output, as described in [BG03]). The resulting construction is

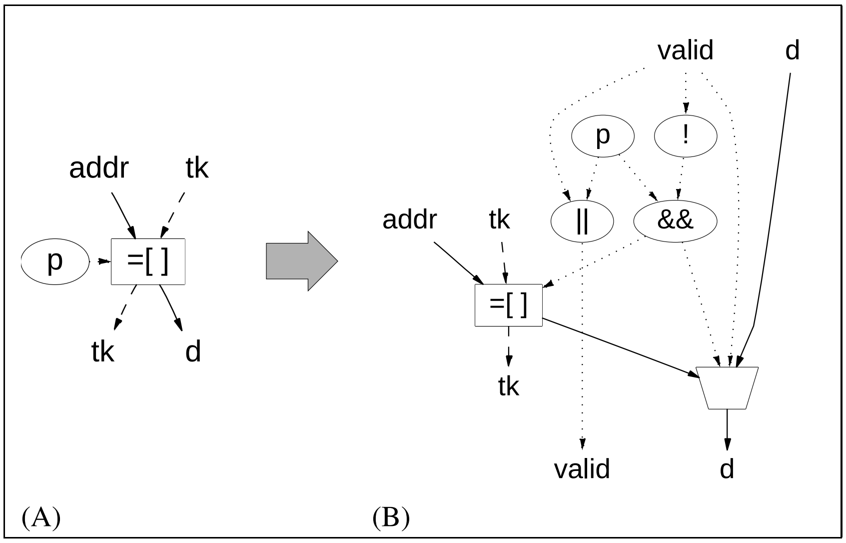

Figure 14: *Term-rewriting of loads for register promotion. "d" and "valid" are the register-promoted data and its valid bit respectively.*

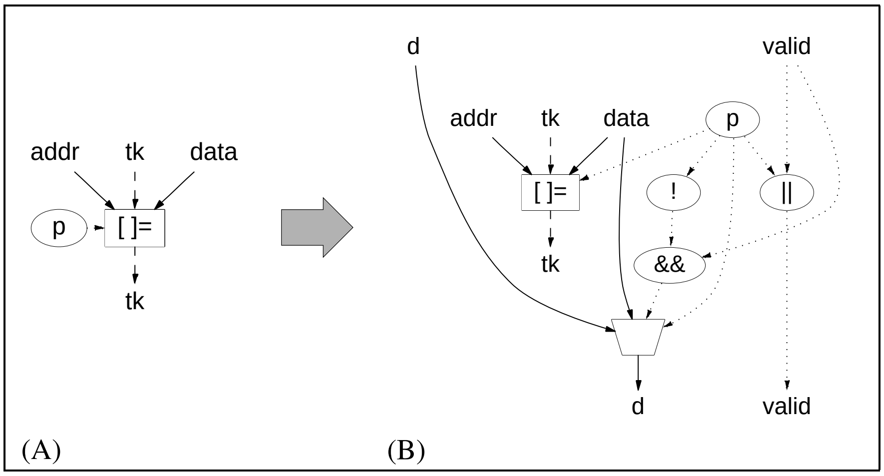

Figure 15: *Term-rewriting of stores for register promotion.*

interpreted as follows: "If the store occurs, the data-to-be-stored replaces the register-promoted data, and it becomes valid. Otherwise, the register-promoted data remains unchanged."

5. Code is synthesized to shift the scalar values and predicates around between strands (the assignments $\mathtt{t}_{j-1}$ = $\mathtt{t}_j$), as illustrated in Figure 16.

6. The insertion of a loop postlude is somewhat more difficult in general than a loop prelude, since by definition natural loops have a unique entry point, but may have multiple exits. In our implementation each loop body is completely predicated and therefore all instructions get executed, albeit some are nullified by the predicates. The compensating stores are added to the loop body and executed only

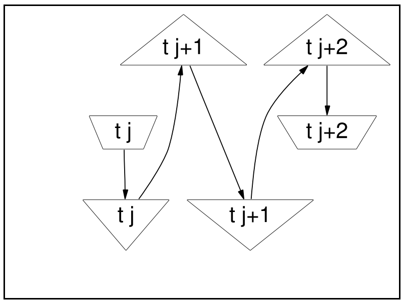

Figure 16: *Shifting scalar values between strands. A similar network is used to shift the "valid" bits.*

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

Figure 17: *Sample code with loop-invariant memory accesses.* `c1` *and* `c2` *stand for loop-invariant expressions.*

during the last iteration. This is achieved by making the predicate controlling these stores to be the loop-termination predicate. This step is not illustrated.

5 Handling Loop-Invariant Predicates

The register promotion algorithm described above can be improved by handling specially loop-invariant predicates. If the disjunction of the predicates guarding all the loads and stores of a same location contains a loop-invariant subexpression, then the initialization load can be lifted out of the loop and guarded by that subexpression. Consider Figure 17 on which we apply loop-invariant scalar-promotion.

By applying our register promotion algorithm one gets the result in Figure 18. However, using the fact that `c1` and `c2` are loop-invariant the code can be optimized as in Figure 19. Both Figure 18 and Figure 19 execute the same number of loads and stores, and therefore, by our optimality criterion, are equally good. However, the code in Figure 19 is obviously superior.

We can generalize this observation: the code can be improved whenever the disjunction of all conditions guarding loads or stores from `*p` is weaker than some loop-invariant expression (even if none of the conditions is itself loop-invariant), such as in Figure 20. In this case the disjunction of all predicates is `f(i) || !f(i)` which is constant "true." Therefore, the load from `*p` can be unconditionally lifted out of the loop as shown in Figure 21.

In general, let us assume that each statement s is controlled by predicate with $P(s)$. 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.

```
/* prelude */
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_valid is known to be true */
        tmp += 1;
        tmp_valid = true;
    }

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

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

    fi = f(i);  /* evaluate f(i) only once */
    /* third load from *p */
    if (fi && !tmp_valid) {
        tmp = *p;
        tmp_valid = true;
    }

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

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

Figure 18: *Optimization of the code in Figure 17 without using the invariance of some predicates.*

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

for (i) {
    /* first load from *p redundant */

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

    /* second load from *p redundant */

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

    fi = f(i);

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

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

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

Figure 19: *Optimization of the code in Figure 17 using the invariance of* `c1` *and* `c2`*.*

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

Figure 20: *Code with no loop-invariant predicates, but with loop-invariant memory accesses.*

3. In prelude initialize $\texttt{t}_j\texttt{valid = }P_j^{inv}$.
4. In prelude initialize $\texttt{t}_j\texttt{ = t}_j\texttt{valid ? a[i}_0\texttt{+}j\texttt{] : 0}$.[8]
5. The predicate guarding each statement s_j is strengthened: $P(s_j) := P(s_j) \wedge \neg P_j^{inv}$.

Our current implementation of this optimization in CASH only lifts out of the loop the disjunction of all predicates which are actually loop-invariant.

6 Discussion

6.1 Dynamic Disambiguation

Our scalar promotion algorithm can be naturally extended to cope with a limited number of memory accesses which cannot be disambiguated at compile time. By combining dynamic memory disambiguation [Nic89] with our scheme to handle conditional control flow, we can apply scalar promotion even when pointer analysis determines that memory references interfere. Consider the example in Figure 22: even though dependence analysis indicates that `p` cannot be promoted since the access to `q` may interfere, the bottom part of the figure shows how register promotion can be applied.

This scheme is an improvement over the one proposed by Sastry [SJ98], which stores to memory all the values held in scalars when entering an un-analyzable code region (which in this case is the region guarded by `f(i)`).

6.2 Hardware support

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

Rotating registers were introduced in the Cydra 5 architecture [DHB89] to support software pipelining. These were used on Itanium for 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

[8] $\texttt{i}_0$ is the initial value of `i` in the loop.

```
tmp = *p;
tmp_valid = true;
for (i) {
    fi = f(i);

    /* first load */
    if (fi & !tmp_valid)
        tmp = *p;

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

    /* second store */
    if (! fi) {
        tmp = 2;
        tmp_valid = true;
    }
}
if (tmp_valid)
    *p = tmp;

-------------------------

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

Figure 21: *Optimization of the code in Figure 20 using the fact that the disjunction of all predicates guarding* `*p` *is loop-invariant (i.e., constant) "true" and the same code after further constant propagation.*

```
for (i) {
   s += *p;
   if (f(i)) *q = 0;
}

-------------------------------------

tmp_valid = false;
for (i) {
   if (!tmp_valid) {
      tmp = *p;
      tmp_valid = true;
   }
   s += tmp;
   if (f(i)) {
      *q = 0;
      if (p == q)
         /* dynamic disambiguation */
         tmp_valid = false;
   }
}
```

Figure 22: *Code with ambiguous dependences and its non-speculative register promotion relying on dynamic disambiguation.*

k values. A software-only solution is to pack the k valid bits into a single integer[9] 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.

6.3 Other Applications of SIDE

This paper introduces the SIDE framework for run-time dataflow evaluation, and presents the register promotion algorithm as a particular instance. Register promotion uses the dynamic evaluation of *availability* and uses predication to remove memory accesses for achieving optimality. SIDE is naturally applied to the *availability* dataflow information, because it is a forward dataflow analysis, and its run-time determination is trivial.

PRE [MR79] is another optimization which uses of availability information which could possibly benefit from the application of SIDE. In particular, *safe* PRE forms (i.e., which never introduce new computations on any path) seem amenable to the use of SIDE. While some forms of PRE, such as lazy code

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

motion [KRS92], are optimal, they do incur small overheads; for example, safety and optimality together require the restructuring of control-flow, for example by splitting some critical CFG edges[10]. A technique such as SIDE could be used on a predicated architecture to trade-off the creation of additional basic blocks against conditionally computing the redundant expression.

The technique used by Bodík et al. in [BG97] can be seen as another application of the SIDE framework, this time for the backwards dataflow problem of dead-code. This application is considerably more difficult, a fact reflected in the complexity of their algorithm.

An interesting question is whether this technique can be applied to other dataflow analyses, and whether its application can produce savings by eliminating computations more expensive than the inserted code.

7 Experimental Evaluation

7.1 Expected Performance Impact

The scalar promotion algorithm presented here is optimal with respect to the number of loads and stores executed. But this does not necessarily correlate with improved performance for four reasons.

First, it uses more registers, to hold the scalar values and flags, and thus may cause more spill code, or interfere with software pipelining.

Second, it 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 them is executed dynamically, all the inserted code is overhead. In practice profiling information and heuristics should be used to select the loops which will most benefit from this transformation.

Third, scalar promotion removes memory accesses which hit in the cache,[11] therefore its benefit appears to be limited. However, in modern architectures L1 cache hits are not always cheap. For example, on the Intel Itanium 2 some L1 cache hits may cost as much as 17 cycles [CL03]. Register promotion trades-off bandwidth to the load-store queue (or the L1 cache) for bandwidth to the register file, which is always bigger.

Fourth, 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.

7.2 Performance Measurements

In this section we present measurements of our register promotion algorithm as implemented in the CASH C compiler. We show static and dynamic data for C programs from three benchmark suites: Mediabench [LPMS97], SpecInt95 [Sta95] and Spec CPU2000 [Sta00].

Our implementation does not use `dirty` bits and therefore is not optimal with respect to the number of stores (it may, in fact, incur additional stores with respect to the original program). However, dirty bits can only save a constant number of stores, independent of the number of iterations. We have considered their overhead unjustified. We only lift loop-invariant predicates to guard the initializer; our implementation can thus optimize Figure 17, but not Figure 20. As a simple heuristic to reduce register pressure, we do not scalarize a value if it is not reused for 3 iterations.

[10] "Critical" here means connecting a basic block with multiple successors to a basic block with multiple predecessors.

[11] Barring conflict misses within the loop.

Program	Variables				Program	Variables			
	Invariant		Strided			Invariant		Strided	
	old	new	old	new		old	new	old	new
adpcm_e	0	0	0	0	099.go	40	53	2	2
adpcm_d	0	0	0	0	124.m88ksim	23	10	1	4
gsm_e	1	1	1	0	129.compress	0	0	1	0
gsm_d	1	1	1	0	130.li	1	0	1	1
epic_e	0	0	0	0	132.ijpeg	5	1	9	5
epic_d	0	0	0	0	134.perl	6	0	0	1
mpeg2_e	1	0	1	0	147.vortex	22	20	1	0
mpeg2_d	4	3	0	0	164.gzip	20	0	1	0
jpeg_e	3	0	7	5	175.vpr	7	2	0	0
jpeg_d	2	1	7	5	176.gcc	11	40	5	2
pegwit_e	6	0	3	1	181.mcf	0	0	0	0
pegwit_d	6	0	3	1	197.parser	20	3	3	5
g721_e	0	0	2	0	254.gap	1	0	18	1
g721_d	0	0	2	0	255.vortex	22	20	1	0
pgp_e	24	1	5	0	256.bzip2	2	2	8	0
pgp_d	24	1	5	0	300.twolf	1	2	0	0
rasta	3	0	2	1					
mesa	44	4	2	0					

Table 2: *How often scalar promotion is applied. "New" indicates additional cases which are enabled by our algorithm. We count the number of different "variables" to which promotion is applied. If we can promote arrays* `a` *and* `b` *in a same loop, we count two variables.*

Table 2 shows how often scalar promotion can be applied. Column 3 shows that our algorithm found many more opportunities for scalar promotion that would not have been found using previous scalar promotion algorithms (however, we do not include here the opportunities discovered by PRE). CASH uses a simple flow-sensitive intra-procedural pointer analysis for dependence analysis.

Figure 23 and Figure 24 show the percentage decrease in the number of loads and stores respectively that result from the application of our register promotion algorithms. The data labeled **PRE** indicate the number of memory operations removed by our straight-line code optimizations only. The data labeled **loop** shows the additional benefit of applying inter-iteration register promotion. We have included both bars since some of the accesses can be eliminated by both algorithms.

The most spectacular results occur for `124.m88ksim`, which has substatial reductions in both loads and stores. Only two functions are responsible for most of the reduction in memory traffic: `alignd` and `loadmem`. Both these functions benefit from a fairly straightforward application of loop-invariant memory access removal. Although `loadmem` contains control-flow, the promoted variable is always accessed unconditionally. The substantial reduction in memory loads in `gsm_e` is also due to register promotion of invariant memory accesses, in the hottest function, `Calculation_of_the_LTP_parameters`. This function contains a very long loop body created using many C macros, which expand to access several constant locations in a local array. The loop body contains control-flow, but all accesses to the small array are unconditional. Finally, the substantial reduction of the number of stores for `rasta` is due to the `FR4TR`

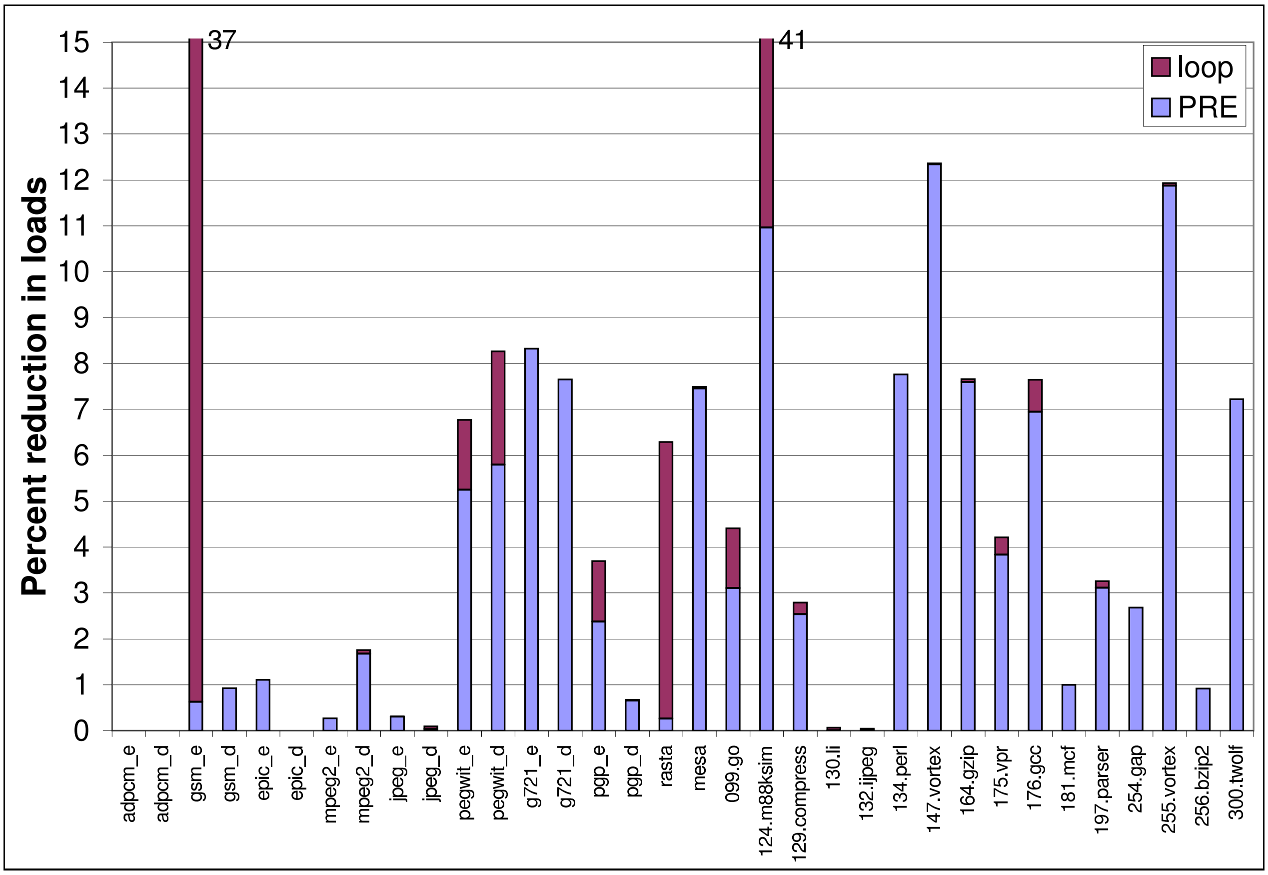

Figure 23: *Percentage reduction in the number of dynamic load operations due to the application of the memory PRE and register promotion optimizations.*

function, which also benefits from unconditional register promotion.

The impact of these reductions on actual execution time depends highly on hardware support. The performance impact modeled on Spatial Computation (described in [BG03, Bud03]) is shown in Figure 25. Spatial Computation can be seen as an approximation for a very wide machine, but which is connected by a bandwidth-limited network to a traditional memory system.

We model a relatively slow memory system, with a 4 cycles L1 cache hit time. Interestingly, the improvement in running time is better if memory is faster (e.g., with a perfect memory system of 2 cycle latency the gsm_e speed-up becomes 18%). This effect occurs because the cost of the removed L1 accesses becomes a smaller *fraction* of total execution cost when memory latency increases.

The speed-ups range from a 1.1% slowdown for 183.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 only 7 programs.

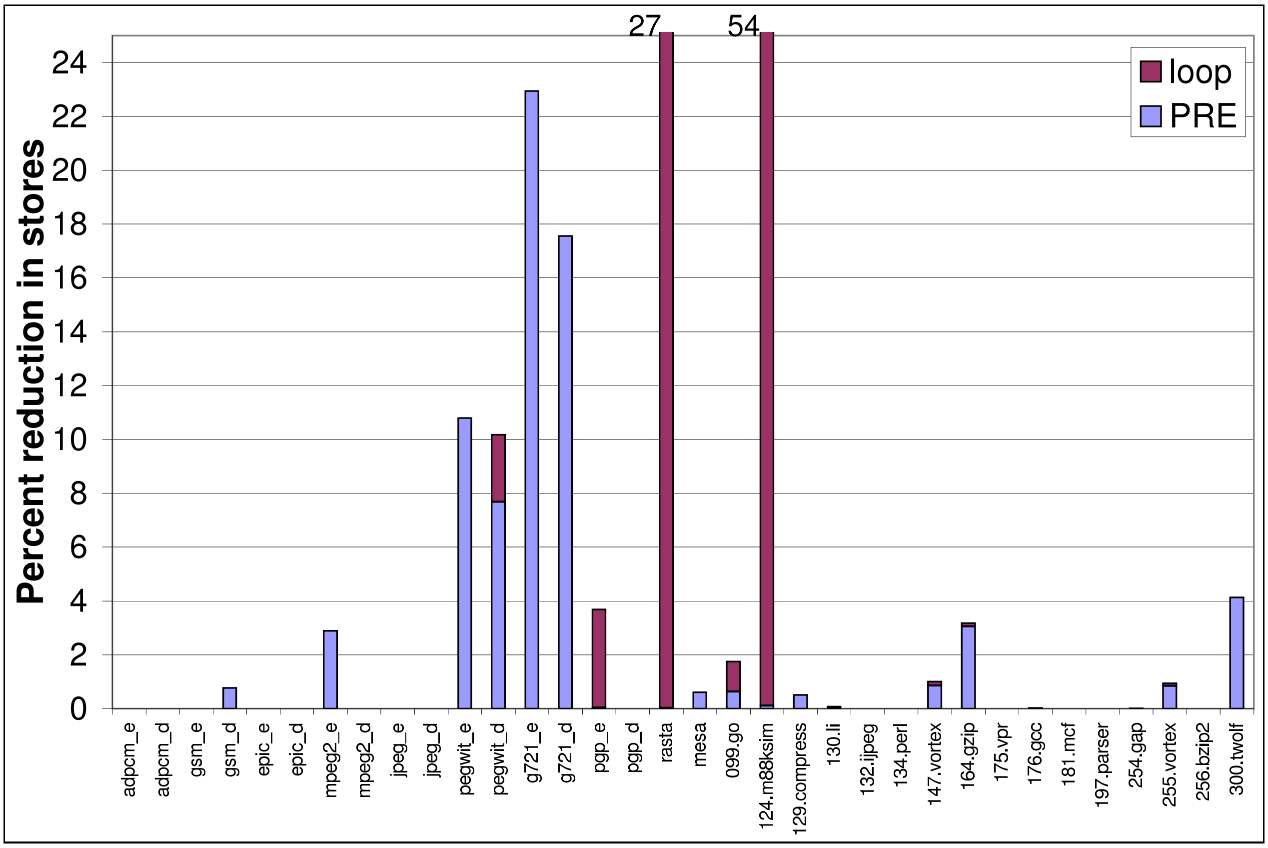

Figure 24: *Percentage reduction in the number of dynamic store operations due to the application of the memory PRE and register promotion optimizations.*

8 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 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.

Schemes that use hardware support for register promotion such as [PGM00, DO94, OG01] are radically different from our proposal, which is software-only. Hybrid solutions, utilizing several of these techniques combined with SIDE, can be devised.

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 also advanced by Bodík [BG97], and was applied for partial dead-code elimination. In fact, the latter paper can be seen as an application of the SIDE framework to the dataflow problem of dead-code. Muchnick [Muc97] gives an example in which

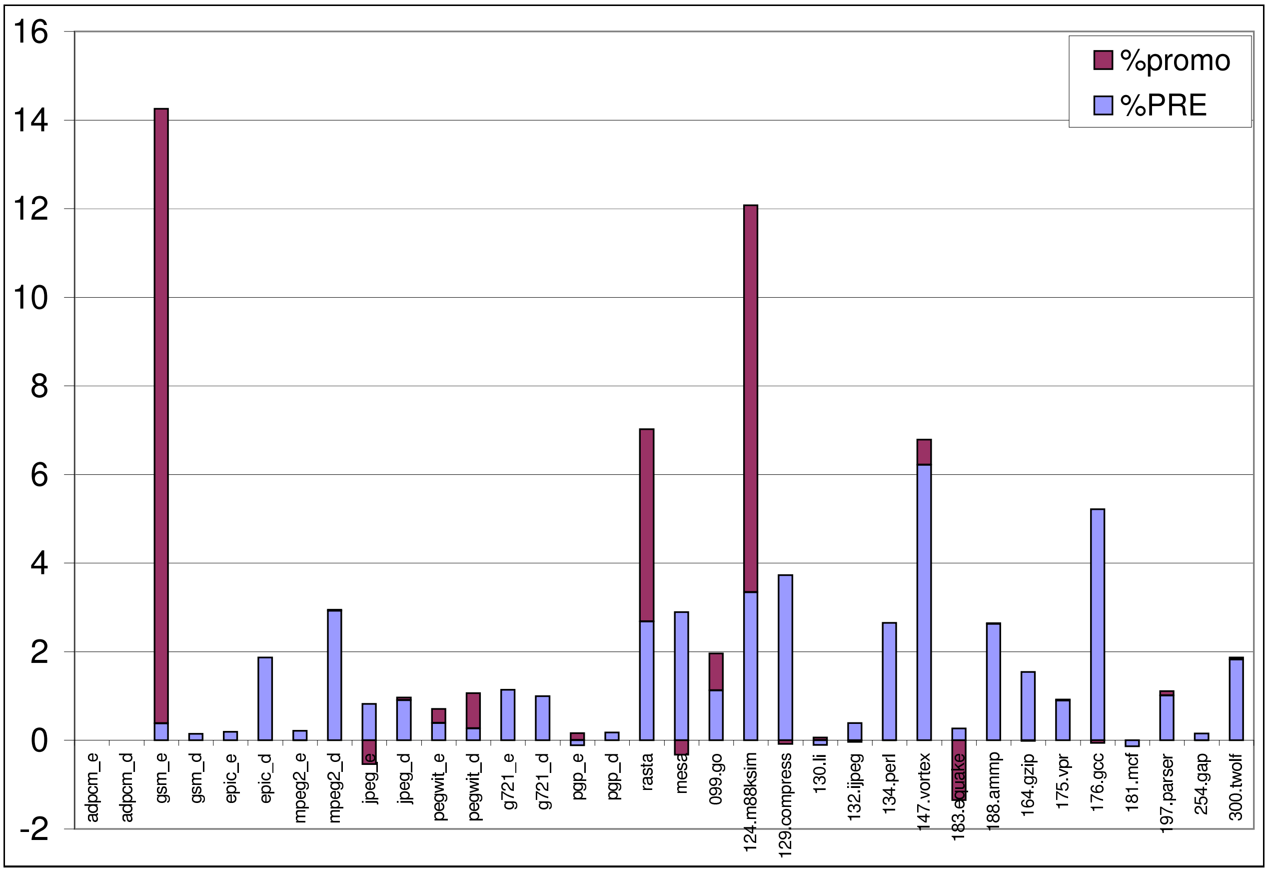

Figure 25: *Percentage reduction in the execution time due to application of our memory optimizations.*

a load can be lifted out of a loop because it occurs on both branches of an `if` statement (which would optimize our Figure 20 directly as shown in Figure 21), but he doesn't describe a general algorithm for solving the problem optimally.

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. 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; since the removed operations tend to be inexpensive (i.e., they hit in the load-store queue or in the L1 cache), the resulting performance improvements are relatively modest.

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.

[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.

[BG02a] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2002.

[BG02b] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[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.

[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.

[Bud03] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[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.

[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.

[CSC$^+$00] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[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.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 224–234. ACM Press, 1992.

[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.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Computer Architecture (ISCA)*, pages 45–54, Dec 1992.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

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

[Nic89] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers (TOC)*, 38 (5):664–678, 1989.

[OBM90] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 257–271, 1990.

[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.

[Sta95] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[Sta00] Standard Performance Evaluation Corp. *SPEC CPU 2000 Benchmark Suite*, 2000. http://www.specbench.org/osg/cpu2000.