Increasing the Applicability of Scalar Replacement

Byoungro So[1] and Mary Hall[2]

[1] IBM T. J. Watson Research Center
1101 Kitchawan Road / Route 134, Yorktown Heights, NY 10598
bso@us.ibm.com

[2] University of Southern California / Information Sciences Institute
4676 Admiralty Way Suite 1001, Marina del Rey CA 90292
mhall@isi.edu

Abstract. This paper describes an algorithm for scalar replacement, which replaces repeated accesses to an array element with a scalar temporary. The element is accessed from a register rather than memory, thereby eliminating unnecessary memory accesses. A previous approach to this problem combines scalar replacement with a loop transformation called unroll-and-jam, whereby outer loops in a nest are unrolled, and the resulting duplicate inner loop bodies are fused together. The effect of unroll-and-jam is to bring opportunities for scalar replacement into inner loop bodies. In this paper, we describe an alternative approach that can exploit reuse opportunities *across multiple loops in a nest*, and *without requiring unroll-and-jam*. We also use this technique to *eliminate unnecessary writes* back to memory. The approach described in this paper is particularly well-suited to architectures with large register files and efficient mechanisms for register-to-register transfer. From our experimental results mapping 5 multimedia kernels to an FPGA platform, assuming 32 registers, we observe a 58 to 90 percent of reduction in memory accesses and speedup 2.34 to 7.31 over original programs.

1 Introduction

A standard strategy for reducing the high (and growing) cost of accessing external memory is to identify multiple memory accesses to the same memory location that *reuse* identical data, and replace unnecessary accesses with references to local storage close to the processor, *e.g.,* registers, caches, compiler-managed buffers. With respect to array variables in loop nest computations, a *data reuse* can be exploited when there are multiple references to an array element in the same or a subsequent iteration of a loop. The first such dynamic reference may either be a read or write; the reuse occurs on subsequent read references. A related optimization opportunity is to eliminate redundant writes to memory, where an array element is written, and on a subsequent iteration is overwritten; only the final value must be written back to memory. In this paper, we refer to this optimization as *register reuse.*

One specific code transformation for exploiting data reuse is *scalar replacement* [1], which replaces array references with scalar temporaries. As such, two references that access the same memory location will instead use the scalar temporaries. This idea can be extended to exploit register reuse by replacing all but the final write of a datum with writes to a register.

One previous approach to scalar replacement eliminates unnecessary read accesses based on true and input dependences in the innermost loop of a nest [2]. It is performed in conjunction with a transformation called *unroll-and-jam*, whereby outer loops in a nest are unrolled, and the resulting duplicate inner loop bodies are fused together. Unroll-and-jam can be used to decrease the distance in the loop nest's iteration space between two dependent references, and increase data reuse in the innermost loop. While this overall strategy has been shown to be very effective at speeding up important computational kernels, the reliance on unroll-and-jam has several limitations. Unroll-and-jam is not always legal, and because it significantly transforms the code, leading to larger loop bodies, it may conflict with other optimization goals.

In this paper, we describe a new approach for scalar replacement of array variables that extends this prior work in several important ways.

1. Increases the applicability by eliminating the necessity for unroll-and-jam;
2. Increases the candidates for reuse both by exploiting reuse across multiple loops in a nest (not just the innermost loop) and by removing redundant write memory accesses (in addition to redundant reads);
3. Provides a flexible strategy to trade off between exploiting reuse opportunities and reducing the register requirements of scalar replacement.

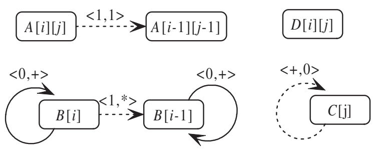
This new approach was motivated by the opportunities arising in compiling to FPGAs: (1) the amount of logic that can be configured for registers far exceeds the number of registers in a conventional system (on the order of several hundred to a few thousand); and, (2) data can be copied between registers in parallel. For these reasons, the compiler stores reused data in registers across loops in a nest, even though the reuse distance can be potentially large, and frequently copies between registers. Further, it is beneficial to avoid the code size increase associated with unroll-and-jam, since increased design complexity can lead to slower achievable clock rates and may exceed device capacity.

While FPGAs represent a unique computing platform, these extensions to scalar replacement will become increasingly important due to a number of current architectural trends, such as the larger register files of 64-bit machines (such as the Itanium's 128 registers), rotating register files [3], and software-managed on-chip buffers (e.g., Imagine [4]). The algorithm described in this paper has been fully implemented in the DEFACTO system, a design environment for mapping applications written in C to FPGA-based systems [5]. We present results on five multimedia kernels, demonstrating a 58 to 90 percent reduction in memory accesses and a speedup of 2.34 to 7.32 over the original programs.

The remainder of this paper is organized as follows. The next section presents background and motivation for this research and briefly introduces the prior solution by Carr and Kennedy [2] for purposes of comparison with our approach.

```
for (i = 1; i < 65; i++)
  for (j = 1; j < 33; j++)
    A[i][j]=A[i-1][j-1]+B[i]+
      B[i-1]+C[j]+D[i][j];
```

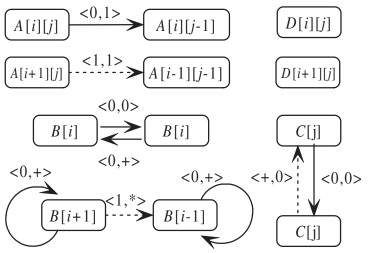
(a) Original code



(b) Reuse graph

```
for (i = 1; i < 65; i+=2)
  for (j = 1; j < 33; j++)
    A[i][j]=A[i-1][j-1]+B[i]+
      B[i-1]+C[j]+D[i][j];
    A[i+1][j]=A[i][j−1]+B[i+1]+
      B[i]+C[j]+D[i+1][j];
```

(c) After unroll-and-jam of loop i



(d) Reuse graph after unroll-and-jam

Fig. 1. An example code

In Section 3, we describe the details of our algorithm, followed by a discussion of code generation in Section 4. Experimental results are presented in Section 5. Related research is discussed in Section 6, followed by a conclusion.

2 Background and Motivation

We use an example code in Fig. 1(a) throughout this paper to illustrate our approach. The write reference $A[i][j]$ and the read reference $A[i\text{-}1][j\text{-}1]$ in Fig. 1(a) represent a data reuse opportunity. The data written by $A[i][j]$ will be read by $A[i\text{-}1][j\text{-}1]$ in the next iteration of loop i and loop j. Similarly, the data fetched by $B[i]$ is read again by $B[i\text{-}1]$ in the next iteration of loop i. Both sets of references carry other data reuse opportunities in that the same data is accessed repeatedly in loop j. The read reference $C[j]$ can be reused repeatedly during the outer loop execution by keeping all portions of array C accessed by the inner loop in a series of registers, provided that there are enough registers available. Array reference $D[i][j]$ does not carry any data reuse opportunity.

A *dependence vector* $\boldsymbol{d} = \langle d_1, d_2, \cdots, d_n \rangle$ refers to a vector difference of distance in an n-dimensional loop iteration space [6]. If the first non-zero entity of a dependence vector is positive, we call it *lexicographically positive*. Fig. 1(b) shows an example of a lexicographically positive dependence. We use D to refer to a set of dependence vectors.

For data reuse analysis, we only consider true, input, and output dependences. Anti-dependences are not considered as candidates of data reuse and are ignored. A dependence vector must also be lexicographically positive for the data reuse to be realizable. Under these requirements, a dependence vector entity which represents the difference in iteration counts between source and sink references of the dependence edge is one of the following:

- c: an integer constant distance.
- $+$: a set of distances that range from 1 to ∞.
- $*$: a set of distances that range from $-\infty$ to ∞, or an unknown distance.

A constant dependence vector entity c means that the distance between two dependent array references in the corresponding loop is c. A *loop-independent* data dependence whose vector entities are all zeros occurs when an array element is referenced multiple times in the same loop iteration. Since we are considering only lexicographically positive dependences, all dependence vectors must have as their first non-zero entity a positive integer, or '+'.

Just like a dependence graph in dependence analysis, we represent a set of data reuse relations by a directed graph called a *data reuse graph*, which includes a set of reuse instances. A *reuse instance* consists of two dependent array references, one source and one sink, and a *reuse edge* which represents the representative dependence vector between the two array references. A reuse instance is referred to in the literature as *temporal reuse.* Temporal reuse from an array reference to itself is referred to as *self-temporal*, while temporal reuse among different array references is called *group-temporal.* A self-loop reuse edge $\circlearrowright$ in Fig. 1(b) represents self-temporal reuse across loop iterations. Array references with no incoming true or input dependence edges are called *reuse generators.* Array references with no outgoing output dependence edges are called *finalizers.* In Fig. 1(b), $A[i][j]$ in reuse edge $\{A[i][j] \rightarrow A[i\text{-}1][j\text{-}1]\}$ is both the reuse generator and the finalizer.

We briefly introduce a previous approach to scalar replacement by Carr and Kennedy [2] for the purposes of comparison with our approach. As stated earlier, their approach relies on unroll-and-jam to shorten reuse distance and thus reduce the number of registers required to exploit a reuse instance. (It is also used to increase instruction-level parallelism.) Carr and Kennedy's approach restricts scalar replacement to array references with dependences carried by the innermost loop, thus of the form $\langle 0, 0, \cdots\ 0, d_n \rangle$. Further, each dependence must be *consistent*, which means that the dependence distance for the loop that carries the dependence is constant throughout the loop [6]. Unroll-and-jam is used to bring dependences carried by outer loops into the innermost loop. In Fig. 1(c), for example, the array references $A[i\text{-}1][j\text{-}1]$, $B[i\text{-}1]$, and $C[j]$ that induce the dependence vectors $\langle 1, 1 \rangle, \langle 1, * \rangle, \langle +, 0 \rangle$, identified by dotted arrows, are not of the form $\langle 0, 0, \cdots\ 0, d_n \rangle$. After unroll-and-jam, as in Fig. 1(c), some dependences are $\langle 0, 1 \rangle, \langle 0, 0 \rangle, \langle 0, + \rangle$ as illustrated in Fig. 1(d), increasing the innermost loop reuse. Similarly, the dependence $\langle +, 0 \rangle$ of array reference $C[j]$ in Fig. 1(b) is not carried by the innermost loop. By unrolling the outer loop, it creates a loop-independent dependence $\langle 0, 0 \rangle$.

The overall algorithm is to reduce unnecessary memory accesses and to match memory and floating point operations with the peak performance of the target architecture [2]. In this paper, $\boldsymbol{M}$ refers to the number of memory accesses remaining after scalar replacement, and $\boldsymbol{R}$ the number of registers required to exploit data reuse. As unroll factors increase, data reuse increases monotonically. However, R also increase and can exceed the number of available registers. Thus, the algorithm attempts to derive the best unroll factors computed as a function of M and the number of floating point operations under the constraint that R is less than the total number of available registers for a given architecture. Thus, the core of their analysis is to compute M and R, parameterized by the unroll factors $u_1, \cdots, u_{n-1}$ for loops 1 to n-1; the algorithm never unrolls the innermost loop because doing so does not affect the data access patterns.

However, their approach still cannot exploit some data reuse opportunities that the outer loop carries even after unroll-and-jam. Thus, array C and B in Fig. 1(c) repeatedly accesses the same array element across outer loops. Further, unroll-and-jam is not always safe, and may conflict with other compiler optimizations.

Compared to Carr and Kennedy's approach, our approach does not require any further unrolling to expose data locality to the innermost loop. In addition, our approach does not have the limitation of applicable dependences of the form $\langle 0, 0, \cdots\ 0, d_n \rangle$. We show how we achieve these goals in the next section.

3 Extending Scalar Replacement

Since we do not perform unroll-and-jam, a key distinction in our approach is a strategy for analyzing reuse distance across multiple loops in a nest. In conjunction with the analysis, we perform register rotates and shifts, as discussed in the next section, to exploit this reuse. Another key difference is the use of reuse chains, rather than pairwise dependences, to identify and group together a collection of references that may refer to the same location. The effect of a reuse chain is that the number of registers needed to exploit the reuse can be reduced as compared to considering reuse among pairs of references. We consider output dependences, not just input and true dependences. As a consequence of this and the goal of exploiting reuse across multiple loops in a nest, the way in which we partition references is also different.

The steps of our algorithm are the following:

1. Perform the dependence analysis for a given code.
2. Build a reuse graph, where we partition all array references into reuse chains $C^{\emptyset}, C^{c}, C^{i\emptyset}$, and C^{ic}, as described below.
3. Compute R and M for each reuse chain. We compute M as part of register pressure control, as described in Section 3.3.
4. If needed to control register pressure, compute efficiency metric of each reuse chain, and tile some inner loops to tradeoff some data reuse opportunities with a smaller number of registers.
5. Replace array references with appropriate scalars.

6. Insert register shift/rotation statements to transfer a data item from the generator down to other references in a reuse chain.
7. Perform loop peeling and loop-invariant code motion to initialize or finalize registers for loop-invariant array references.

3.1 Definitions

A *reuse chain*, denoted as $\boldsymbol{C}$, is a connected component of a reuse graph. A reuse generator provides the data that is used by all the references in the reuse chain, and the finalizer finalizes the value of the array element that may have been overwritten several times by previous write references.

Since we are exploiting data reuse across multiple loops in a nest, we need a concept of the distance between the references to the same array element across outer loop iterations. A *group temporal reuse distance* is defined by the minimal difference in iteration counts of the innermost loop between two dependent references. A reuse distance is different from a dependence distance in that it represents only the smallest dependence distance between the two dependent array references when there are multiple distances. For a given $\boldsymbol{d} = \langle c, \cdots, c\rangle$ in an n-deep loop nest, we define the group temporal reuse distance $\epsilon(\boldsymbol{d}, n)$ as follows:

$$\epsilon(\boldsymbol{d}, n) = \sum_{l=1}^{n-1} \left\{ \left(\prod_{k=l+1}^{n} I_k \right) \times d_l \right\} + d_n, \tag{1}$$

where I_k is the iteration count of the k-th loop in a nest, and c is an integer constant. In Fig. 1(b), for instance, the reuse distance of a dependence vector $\langle 1, 1\rangle$ of $\{A[i][j] \rightarrow A[i\text{-}1][j\text{-}1]\}$ is 33 (=1×32+1).

We partition reuse chains that exploit different kinds of data reuse across loop iterations into four categories; *i.e.*,

– $\boldsymbol{C^{\emptyset}}$: reuse chains that carry no data reuse opportunities.
– $\boldsymbol{C^{c}}$: reuse chains that carry only group-temporal reuse opportunities.
– $\boldsymbol{C^{i\emptyset}}$: reuse chains that carry only self-temporal reuse opportunities.
– $\boldsymbol{C^{ic}}$: reuse chains that carry both self-temporal and group-temporal reuse opportunities.

Reuse chain categories $C^{\emptyset}$ and C^{c} can contain a loop independent data reuse among lexically identical array references in addition to the conditions above.

Our categorization is different from Carr's in two ways. First, since we are exploiting data reuse across outer loops as well as the innermost loop, we discriminate reuse chains that exploit data reuse only in invariant loops (category $C^{i\emptyset}$) from reuse chains that exploit data reuse in the invariant loops as well as with a constant dependence distance in some loops (category C^{ic}). Secondly, we categorize both read and write references, which allows us to eliminate redundant writes back to memory. Table 1 shows the characteristics of the dependences in each category and examples of reuse chains from Fig. 1(b). Note that we compute M and R throughout the entire execution of a loop nest.

Table 1. Comparison of reuse chain categories.

	$C^{\emptyset}$	C^{c}	$C^{i\emptyset}$	C^{ic}
D	$\emptyset$ or $\{0,\cdots,0\}$	$\{\langle c,\cdots,c\rangle\}$	$\{\langle\cdots,+/*,\cdots,0,\cdots\rangle\}$	$\{\langle\cdots,+/*,\cdots,c,\cdots\rangle\}$
e.g.	$\{D[i][j]\}$	$\{A[i][j] \rightarrow$ $A[i\text{-}1][j\text{-}1]\}$	$\{C[j] \circlearrowleft\}$	$\{\circlearrowright B[i] \rightarrow B[i\text{-}1]\circlearrowleft\}$
G	2,048	2,048	32	64
A		1,953		63
M	2,048	2,143	32	65
R	0	34	32	2

In the next section, we use the following terms to describe the algorithm. ***A*** refers to the group-temporal reuse associated with a particular incoming dependence or reuse chain; *i.e.*, how many memory accesses associated with one or more dependences can be eliminated. ***G*** refers to the number of memory accesses incurred, associated with a reuse generator. In addition, we assume that there are enough registers to exploit all the data reuse opportunities exposed by data reuse analysis, and that the loop bounds are constant. We relax these assumptions by generalizing the algorithm in Section 3.3.

3.2 Computing *R* and *M* for Each Reuse Chain

We compute the number of registers and the number of memory accesses for each reuse chain, while Carr's approach does so for each individual array reference. Because the data of a reuse generator may be used several times until the last reference in a reuse chain uses it, the dependence vector $\boldsymbol{d}$ between the generator and the last reference in a reuse chain decides the required number of registers to fully exploit possible data reuse.

For a given $\boldsymbol{d}$ and a set of loop iteration counts, we compute A and G for each data reuse chain as follows:

$$G = \prod_{l=1}^{n} \alpha(d_l, I_l), \text{ where } \alpha(d_l, x) = \begin{cases} x, \text{ if } d_l \text{ is a constant;} \\ 1, \text{ otherwise,} \end{cases} \tag{2}$$

$$A = \prod_{l=1}^{n} \beta(d_l, I_l), \text{ where } \beta(d_l, x) = \begin{cases} 1, & \text{if } d_l \text{ is not a constant;} \\ x - d_l, & \text{if } x \geq d_l; \\ 0, & \text{if } x < d_l. \end{cases} \tag{3}$$

The above equations are functions of I rather than unroll factors, since we exploit data reuse in the entire loop nest and do not unroll the loops. In Table 1, for example, G=2,048 and A=1,953 for reuse chain $\{A[i][j] \overset{\langle 1,1\rangle}{\rightarrow} A[i\text{-}1][j\text{-}1]\}$.

In the rest of this subsection, we describe how to compute R and M for each type of reuse chain for a given dependence vector $\boldsymbol{d} = \langle d_1, d_2, \cdots, d_n\rangle$ between the generator and the last array reference in the reuse chain.

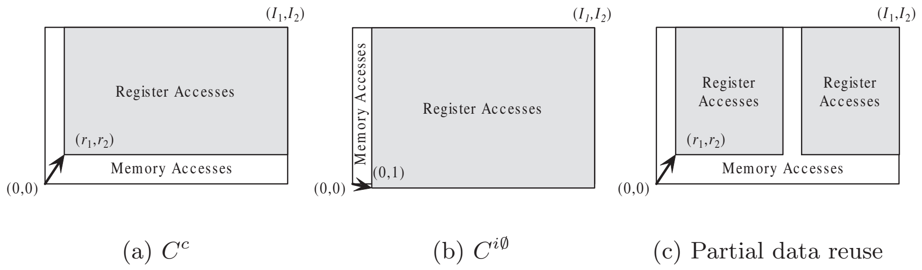


Fig. 2. Memory/register access behaviors of full and partial data reuse.

Reuse chains in $C^{\emptyset}$ carry no reuse across iterations, but multiple references in the chain may carry loop-independent reuse within the loop body. In this case, we use one register to exploit loop-independent data reuse from the generator. Thus, M and R for reuse chains in $C^{\emptyset}$ can be computed as follows:

$$M^{\emptyset} = G, \qquad R^{\emptyset} = \begin{cases} 0, & \text{if } d = \emptyset \\ 1, & \text{if } d = \langle 0, \cdots, 0 \rangle. \end{cases}$$

In Table 1, for example, $M^{\emptyset}$=2,048 and $R^{\emptyset}$=0 for reuse chain $\{D[i][j]\}$, since it does not exploit any data reuse.

Reuse chains in C^c captures only group-temporal reuse. Therefore, M and R required for reuse chains in C^c can be computed as follows:

$$M^c = 2G - A, \qquad R^c = \epsilon(\boldsymbol{d}, n) + 1,$$

where the function $\epsilon(\boldsymbol{d}, n)$ is defined in Equation 1.

The reuse generator accesses memory in the entire iteration space, which leads to G memory accesses. Other references in the same reuse chain fetch data from memory only in the initial d_l iterations of each loop l. Figure 2(a) illustrates the general data access behavior for a reuse chain with reuse distance $\langle d_1, d_2 \rangle$. In the shaded region, which represents the group-temporal reuse amount A, the sink reference of a reuse instance gets necessary data from the source reference. However, the sink references access memory in the L-shaped white region. To exploit data reuse for reuse chains in C^c, a series of registers are necessary to keep the data until it is fully reused. The reuse generator fetches data from memory to the first register in a series, and each register A_i shifts the data to its neighboring register A_{i+1} in every innermost loop's iteration. Finally, after iterations of the innermost loop corresponding to the reuse distance $\epsilon(\boldsymbol{d}, n)$, the last register $A_{\epsilon(\boldsymbol{d},n)}$ gets the necessary data. Meanwhile, the reuse generator keeps fetching a new datum from memory in every iteration; i.e., G memory accesses. The L-shaped white region can be computed by subtracting the shaded region from the whole region; *i.e.*, $G - A$.

Therefore, the total number of memory accesses for a reuse chain in C^c is $G + G - A = 2G - A$, since the generators as well as other references belong to C^c.In Table 1, for example, M^c=2,143 and R^c=34 for reuse chain $\{A[i][j] \overset{\langle 1,1\rangle}{\rightarrow} A[i\text{-}1][j\text{-}1]\}$.

Reuse chains in $C^{i\emptyset}$ have a dependence vector that consists of '+', '$*$', and 0. Let d_j be the first non-zero dependence distance, which should be '+' in this reuse chain category; *i.e.*, $\langle 0, \cdots, 0, +, \cdots \rangle$. Then, M and R required for each reuse chain in $C^{i\emptyset}$ can be computed as follows:

$$M^{i\emptyset} = G, \qquad R^{i\emptyset} = \prod_{l=j}^{n} \alpha(d_l, I_l), \text{ where } d_j = \text{'+'}.$$

The data in a register can be reused without any additional registers during the entire execution of loops corresponding to '+' or '$*$'.Figure 2(b) illustrates the general data access behavior for a reuse chain with dependence vector $\langle 0, + \rangle$. Array elements along the dimensions corresponding to the dependence distance 0 are kept in a series of registers for the outer loops corresponding to dependence distance '+' or '$*$' to exploit self-temporal reuse. In the dimensions corresponding to leading zero dependence distances (*i.e.*, dimensions from 1 to j-1 if $j > 1$), the reuse chain does not exploit data reuse at all, so they do not affect the number of registers. In Table 1, for example, $M^{i\emptyset}$=32 and $R^{i\emptyset}$=32 for reuse chain $\{C[j] \circlearrowleft^{\langle +,0\rangle}\}$.

Reuse chains in C^{ic} contain self-loop edges and data reuse edges between different array references. As such, a dependence vector of reuse chains in C^{ic} consists of '+', '$*$', and constants. In this category, at least one constant should be non-zero.

Let d_j be the first non-constant dependence distance; *i.e.*, $\langle c, \cdots, c, +/*, \cdots \rangle$. Then, the number of memory accesses remaining after scalar replacement for reuse chains in C^{ic} can be computed as follows:

$$M^{ic} = 2G - A, \qquad R^{ic} = \{\epsilon(\boldsymbol{d}, j-1) + 1\} \times \prod_{l=j}^{n} \alpha(d_l, I_l).$$

In Table 1, for example, reuse chain $\{\circlearrowleft B[i] \overset{\langle 1,*\rangle}{\rightarrow} B[i\text{-}1]\circlearrowleft\}$ can exploit self-temporal data reuse in the inner loop (corresponding to '$*'$) and group-temporal data reuse in the outer loop (corresponding to '1'). Since two references are invariant in the inner loop, we can determine the reuse distance in terms of the outer loop, which is 1. Thus, R^{ic}=1+1=2 for this chain.

Consider dependence vector $\langle +, 1 \rangle$. A data item is reused in the next iteration of the inner loop. In addition, the data is reused during the entire iterations of the outer loop. Thus, the same number of registers as the inner loop iteration count are necessary, no matter how small c is. Therefore, the first non-constant dependence distance in dimension j means that exploiting self-temporal reuse in j through n dimensions requires the following number of registers just like $R^{i\emptyset}$; *i.e.*, $\prod_{l=j}^{n} \alpha(d_l, I_l)$.

```
for ($i$ = 1; $i$ < 65; $i$++) {
   $B_0 = B[i]$;
   if ($i$==1)   $B_1 = B[i\text{-}1]$;
   for ($j$ = 1; $j$ < 33; $j$++) {
      if ($i$==1)
         { $A_{33} = A[i\text{-}1][j\text{-}1]$;   $C_0 = C[j]$; }
      else if ($j$==1)   $A_{33} = A[i\text{-}1][j\text{-}1]$;
      $A_0 = A_{33} + B_0 + B_1 + C_0 + D[i][j]$;
      $A[i][j] = A_0$;
      shift_registers($A_0, \cdots, A_{33}$);
      rotate_registers($C_0, \cdots, C_{31}$);
   }
   $B_1 = B_0$;
}
```

(a) Full data reuse

```
for ($j_t$ = 1; $j_t$ < 33; $j_t$+=16)
   for ($i$ = 1; $i$ < 65; $i$++) {
      $B_0 = B[i]$;
      if ($i$==1)   $B_1 = B[i\text{-}1]$;
      for ($j = j_t$; $j < j_t$+16; $j$++) {
         if ($i$==1)
            { $A_{17} = A[i\text{-}1][j\text{-}1]$;   $C_0 = C[j]$; }
         else if ($j$==$j_t$)   $A_{17} = A[i\text{-}1][j\text{-}1]$;
         $A_0 = A_{17} + B_0 + B_1 + C_0 + D[i][j]$;
         $A[i][j] = A_0$;
         shift_registers($A_0, \cdots, A_{17}$);
         rotate_registers($C_0, \cdots, C_{15}$);
      }
      $B_1 = B_0$;
   }
```

(b) Partial data reuse

Fig. 3. Final output of our approach to scalar replacement.

Further, the constant dependence vector dimensions in the outer loops before dimension j require $\epsilon(\boldsymbol{d}, j\text{-}1)+1$ sets $R^{i\emptyset}$ registers in a similar way as computing R^c. For example, consider a dependence vector $\langle 1, *, 2\rangle$ between two references $A[i][k]$ and $A[i\text{-}1][k\text{-}2]$ within a 3-deep loop nest. To exploit self-temporal reuse in the second dimension, I_k registers are required to keep the array elements accessed in the innermost loop. Further, since the dependence distance in the first dimension is 1, the total number of registers required for this reuse chain ends up multiplying 2 by I_k. Intuitively, this corresponds to I_k registers to exploit self-temporal reuse in the second dimension, and an additional I_k registers to exploit group-temporal reuse in the first dimension. Thus, G=64, A=63, M^{ic}=65, and R^{ic}=2 for reuse chain $\{\circlearrowright B[i] \overset{\langle 1,*\rangle}{\rightarrow} B[i\text{-}1]\circlearrowleft\}$.

Fig. 3(a) shows the result of our scalar replacement applied to the code in Fig. 1(a). The code in Fig. 3(a) does not access any array element more than once, fully exploiting data reuse opportunities at the cost of more registers used than Carr's approach. In the next subsection, we present how we control the register pressure, not losing many opportunities of data reuse.

3.3 Generalizing the Algorithm

In architectures where there are not enough registers to exploit all possible data reuse opportunities, we have to reduce the required number of registers to avoid register spills. Further, a non-constant loop bound hinders the reuse analysis from computing the reuse distance $\epsilon(d, n)$ in Equation 1. Fortunately, we can use the same technique to deal with these issues.

Partial data reuse trades off data reuse opportunities for lower register pressure. We exploit partial data reuse using a code transformation called *tiling*, which divides computation and data into blocks, and makes the data reuse closer in time. Partial data reuse exploits data reuse only within a tile and introduces memory accesses across tiles. Fig. 2(c) shows the difference in memory access behaviors between full and partial data reuse. The loop to be tiled and the tiling factors are decided based on the dependence information. Tiling each loop i by T_i will reduce the reuse distance by

$$\sum_{j=1}^{n-1}\left\{\left(\prod_{k=j+1}^{n} I_k\right)\times d_j\right\} - \sum_{j=1}^{n-1}\left\{\left(\prod_{k=j+1}^{n} T_k\right)\times d_j\right\}.$$

¿From this tiled reuse distance, we can compute the number of registers and the number of memory accesses required for the tiled code. Fig. 3(b) shows the code after tiling the inner loop j, which requires 36 registers for partial data reuse and incurs 4,416 memory accesses.

Using tiling, we can also derive inner loop nests with constant loop bounds, when some bounds are not constant. If the iteration count is not divisible by the tiling factor, we can use *Index set splitting*, which divides the index set of a loop into multiple portions, replicating the body of the loop as appropriate [6]. We use index set splitting to isolate the residual (I_k `mod` T_k) iterations from the main tiled loop l_k. As a result, each tile of iterations contains a fixed constant T_k iteration count.

Tiling is not always a legal transformation[1]. An alternative way to control register pressure is *selective data reuse* which selectively exploits data reuse for some reuse chains depending on their register requirements and reuse benefits. Thus, we could alternatively compute the register efficiency for each reuse chain c as follows:

$$\texttt{Efficiency}_c = \overline{M}_c / R_c,$$

where $\overline{M}_c$ refers to the number of memory accesses eliminated by scalar replacement and R_c the number of registers necessary for reuse chain c. A hybrid approach could apply selective data reuse to filter out extremely low efficiency reuse chains, and apply partial data reuse for the rest.

4 Code Transformation

Once the number of registers is computed, a series of scalar variables, $S_0, S_1, \cdots$, S_{R-1}, are introduced, where R is the number of registers to be used to replace array references in a reuse chain. An array reference is replaced with an appropriate scalar variable depending on the reuse distance between its generator and the array reference within the same reuse chain. Each reuse generator that is a read reference needs a register initialization "S_i=A[ ]" before the register is used, and the finalizer reference in a reuse chain requires a memory store "A[ ]=S_i". Intuitively, the insertion position of these initialization/finalization statements is the innermost loop in which the array reference is not invariant, thereby accomplishing loop-invariant code motion. A discussion of control flow is beyond the scope of this paper, but modifications to support control flow are described in [7].

[1] Its legality test is equivalent to unroll-and-jam.

4.1 Register Shift / Rotation

A reference receives data from its generator after the number of iterations corresponding to its reuse distance. Let r represent the reuse distance between the generator and the last array reference in a reuse chain. Then, to achieve reuse, a `shift_registers` operation, equivalent to a series of register copy statements $\{(S_{i+1} = S_i) \mid r-1 \geq i \geq 0\}$, is inserted at the appropriate position as in Fig. 3. The insertion position of register shifts for each reuse chain can be decided by the same method as is used to determine the initialization/finalization point of registers.

In the case of reuse chain categories $C^{i\emptyset}$ and C^{ic}, data reuse occurs repeatedly in loops where the array reference is invariant. The `rotate_registers` operation shifts the data in a series of registers and rotates the last one into the first position.

4.2 Loop Peeling and Code Motion

Loop peeling removes the first (or last) several iterations of the loop body and makes them as separate code before (or after) loop execution. The scalar replacement optimization uses loop peeling in combination with loop-invariant code motion for register initialization and finalization purposes.

We see in Fig. 3(a) that values for register A_{33} are initialized on the first iteration of loop j and on the first iteration of loop i. For clarity it is not shown here, but the code generated by our compiler actually peels the first iteration of loop j and loop i instead of including these conditional loads so that the main body of both loops have the same number of memory accesses. Further, we can avoid the overhead of condition checks in every loop iteration. In addition, array reference $B[i]$ is invariant with respect to loop j, so the initialization of register B_0 and B_1 are moved outside the inner loop j. Within the main unrolled loop body, only a read reference to array D and a write reference to array A remain.

Our approach performs loop peeling on multiple loops in a nest. To initialize the scalar variables at the beginning of the loop iteration, each loop l_i needs to be peeled by the maximum dependence distance among all true and input dependence vectors. To finalize the array elements at the end of the loop iteration, each loop l_i needs to be peeled by the maximum dependence distance among all output dependence vectors. However, since we peel each loop by the maximal dependence distance among all dependence vectors, some array references whose dependence distance is less than the peel factor do not need to be initialized because they are lexically identical array references in some peeled iterations.

5 Experiment

This section presents experimental results that characterize the impact of the scalar replacement algorithm for a set of application kernels written in C: a digital FIR filter (FIR), matrix multiply (MM), pattern matching (PAT), Jacobi 4-point stencil (JAC), and Sobel edge detection (SOBEL). MM is implemented by a 3-deep loop nest, and the others are 2-deep loop nests.

Table 2. Problem size (iteration count).

	FIR			MM			PAT			JAC			SOBEL		
Problem size	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Outer loop	64	64	640	64	64	640	64	64	640	32	32	320	64	64	640
Middle loop				5	6	6									
Inner loop	30	32	60	5	6	16	31	32	62	15	16	30	14	16	28

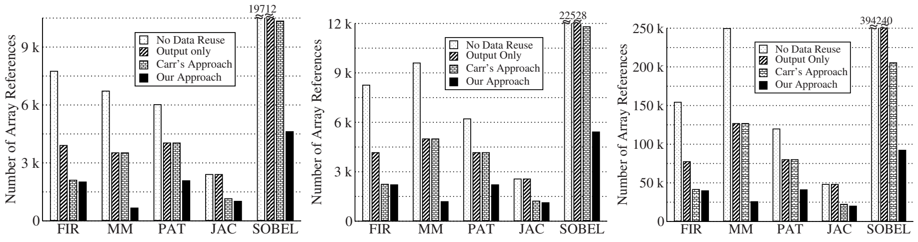
5.1 Methodology

The algorithm described in this paper has been fully implemented in the DEFACTO system, a design environment for mapping C applications to application-specific FPGA hardware implementations [5]. As compared to a conventional architecture, FPGAs have no instruction or data cache, and the microarchitecture is configured specifically for the application. DEFACTO is built on top of the Stanford SUIF compiler, in which we have implemented the scalar replacement transformation. After the compiler applies its transformations, DEFACTO automatically generates a VHDL hardware description of the algorithm, which is then synthesized to hardware by commercially available FPGA synthesis tools; the measurements provided in this section were obtained from Mentor Monet [8], a behavioral synthesis tool. While the output of Monet is not identical to the results obtained from executing the application in hardware, the number of cycles is the same and can be compared directly; other metrics have been shown to be consistent between Monet and the final implementation [9]. The target FPGA device is a Xilinx Virtex XCV 1000 [10].

In this experiment, we compare four data reuse schemes: (1) no data reuse, (2) redundant write elimination only, (3) an approximation to Carr's approach, and (4) our approach. The approximation to Carr's approach, selects the maximal unroll factor for the outer loop of the 2-deep loop nests such that the number of registers is not exceeded. Inner loops are not unrolled in Carr's algorithm. In the case of MM, we also unrolled the outermost loop since the results would be equivalent for either of the two outer loops. While Carr's algorithm might decide not to unroll this much, these results can be thought of as an upper bound on the benefits from Carr's scalar replacement algorithm, since the measurements for performance improvements on our FPGA system are not affected by code size.

To characterize the benefits and cost of each data reuse scheme, we measured four metrics: (1) the number of memory accesses remaining after each data reuse scheme, and (2) the number of registers used to exploit data reuse, and (3) the speedup over original programs, and (4) the FPGA space usage.

The first and second metrics heavily depend on the iteration counts of the loops. For each program, thus, we compare three different problem sizes in terms of iteration count of each loop in a nest as shown in Table 2. We assume there are 32 registers available in the target architecture. The first problem size for each program requires less than or equal to 32 registers to fully exploit data



(a) Problem size 1 (b) Problem size 2 (c) Problem size 3

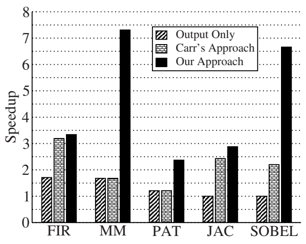
Fig. 4. Number of Memory Accesses.

reuse using our approach. The second size is slightly bigger than the first size, and thus it requires slightly more registers. The third size requires much more registers than 32 to exploit full data reuse. Thus, we perform partial data reuse in the cases of the second and third problem sizes of each program. For Carr's approach, we maximize the unroll factor of the outer loop (and jam the copies of the inner loop) such that it uses less than or equal to 32 registers. Our approach is fully automated, but we derived results for output elimination only and Carr's approach with some manual direction of our compiler.

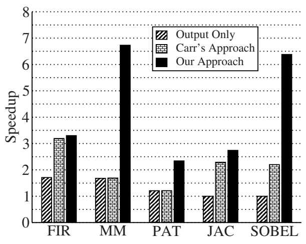
5.2 Experimental Results

In Fig. 4, we measured how effectively we eliminated the quantity of redundant memory accesses, with each bar capturing results for one of the four previously described schemes. In Fig. 4(a) and (b), the second bar eliminated about 50% of memory accesses for FIR and MM, 33% for PAT. However, there was no opportunity for redundant write elimination at all for JAC and SOBEL. In Fig. 4(c), the second bar eliminated slightly more memory accesses (up to 1%), since the problem size is bigger. For three different problem sizes, Carr's approach eliminated 72% of memory accesses for FIR, 52% for JAC, and 47% for SOBEL. However, it eliminated the same amount of redundant memory accesses as the second bar did for MM and PAT. On the other hand, our approach eliminated slightly more redundant memory accesses (up to 5% more) than Carr's approach for FIR and JAC. However, it eliminated 90% of memory accesses for MM, 65% for PAT, and 77% for SOBEL in Fig. 4(a), and up to 3% less in Fig. 4(b), and up to 1% more in Fig. 4(c). If the problem size is bigger, the benefits of our approach to scalar replacement would be greater.

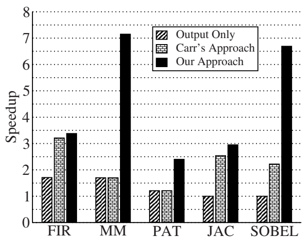
Table 3 shows the number of registers required to exploit data reuse for three data reuse schemes. Carr's approach uses 32 registers for FIR, MM, and PAT, but 30 registers for JAC and SOBEL because increasing the unroll factor requires more than 32 registers. Only one register is required to eliminate redundant write accesses for FIR, MM, and PAT. There are no redundant array writes for JAC or SOBEL. Our approach uses a comparable number of registers for problem sizes 1 and 3, but considerably less registers for problem size 2 because tiling reduces



(a) Problem size 1



(b) Problem size 2



(c) Problem size 3

Fig. 5. Speedups of scalar replacement.

the iteration count of the inner loop. However, as shown in Figure 4(b), partial data reuse eliminated more redundant array accesses than in the (c) case, which uses many more registers.

Fig. 5 presents the speedup results of overall performance on a single FPGA, again corresponding to the four schemes above. The approximation to Carr's approach observes speedups from 1.21 to 3.21, and output elimination only observes speedups from 1.21 to 1.71. On the other hand, our approach observes speedups from 2.34 to 7.31 for five programs. If we use full data reuse, which is possible on the FPGA platform for the smaller problem sizes, the speedups of our approach would be greater.

In summary, from our experiments, we observe several sources of benefits. While most benefits come from exploiting reuse of input and true dependences, redundant write elimination is valuable and reduces the number of memory accesses up to 51%. Exploiting data reuse across multiple loops does not necessarily require a large number of registers relative to the footprint of the accessed data, and results in the best performance. Partial data reuse within a limited number of registers eliminated up to 40% more redundant array accesses than our approximation to Carr's approach. As a final benefit of our approach over using unroll-and-jam, the FPGA space required to implement the unrolled computation may be prohibitive. Larger designs have more routing complexity, and may lead to lower achieved clock rates. For the example kernels using the maximal unroll factors, the designs resulting from our algorithm were up to 80% smaller.

Table 3. Number of registers.

	FIR			MM			PAT			JAC			SOBEL		
Problem size	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Carr's appr.	32	32	32	32	32	32	32	32	32	30	30	30	30	30	30
Output only	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
Our appr.	31	17	31	30	19	29	32	17	32	31	17	31	32	20	32

6 Related Work

The most closely related work [2] is explained in detail in this paper. While there have been numerous other works on data locality optimizations, we cite only those most closely related to our transformations here. Fink et al. [11] introduces a scalar replacement algorithm that can handle both array and pointer objects. In their model, they treat both pointer and array references as accesses to elements of hypothetical heap arrays and they can avoid an expensive point-to analysis. McKinley et al. [12] showed a compound data locality optimization algorithm to improve cache performance that place the loop that carries the most data reuse at the innermost position for each statement in a loop nest. Kolson et al. [13] uses an *internal data forwarding* technique to eliminate redundant memory traffic in a pipelining scheduler. Kodukula et al. [14] introduces *data shackling*, a data-centric approach to locality enhancement for L2 cache. It fixes a block of data and then determines which computations should be executed on it. Deitz et al. [15] introduces *array subexpression elimination* that eliminates redundant computation and accompanying memory accesses.

7 Conclusion

In this paper, we have described an algorithm for scalar replacement, used to eliminate redundant memory accesses by replacing array references with scalar temporaries. Our approach extends scalar replacement (1) to increase the applicability by eliminating the necessity for unroll-and-jam, and (2) to increase the candidates for reuse both by exploiting reuse across multiple loops in a nest (not just the innermost loop) and by removing redundant write memory accesses (in addition to redundant reads), and (3) to provide a flexible strategy to trade off between exploiting reuse opportunities and reducing the register requirements of scalar replacement. Using less than or equal to 32 registers for this technique, we observe a 58 to 90 percent of reduction in memory accesses and speedup of 2.34 to 7.31 over the original programs.

References

1. Callahan, D., Carr, S., Kennedy, K.: Improving register allocation for subscripted variables. In: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, New York, ACM press (1990)
2. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. ACM Transactions on Programming Languages and Systems **16** (1994) 1768–1810
3. Ng, J., Kulkarni, D., Li, W., Cox, R., Bobholz, S.: Inter-procedural loop fusion, array contraction and rotation. In: The 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA (2003) 114–124
4. Rixner, S., Dally, W.J., Kapasi, U.J., Khailany, B., Lopez-Lagunas, A., Mattson, P.R., Owens, J.D.: A bandwidth-efficient architecture for media processing. In: International Symposium on Microarchitecture. (1998) 3–13

5. So, B., Hall, M.W., Diniz, P.C.: A compiler approach to fast hardware design space exploration in fpga-based systems. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02), New York, ACM press (2002) 165–176
6. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco (2002)
7. So, B.: An Efficient Design Space Exploration for Balance between Computation and Memory. Ph.d. dissertation, University of Southern California, Los Angeles, CA (2003)
8. Mentor Graphics Inc.: Monet^{TM}. R44 edn. (1999)
9. So, B., Diniz, P., Hall, M.: Using estimates from behavioral synthesis tools in compiler-directed design space exploration. In: The 40th Design Automation Conference, Anaheim, CA (2003)
10. XILINX: Virtex-II 1.5V FPGA Complete Data Sheet. DS031(v1.7), 2100 Logic Drive, San Jose, Calif. (2001)
11. Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: Proceedings of the 2000 Static Analysis Symposium. (2000) 155–174
12. Mckinley, K.S., Carr, S., wen Tseng, C.: Improving data locality with loop transformations. ACM Transactions on Programming Languages and Systems **18** (1996) 424–453
13. Kolson, D., Nicolau, A., Dutt, N.: Elimination of redundant memory traffic in high-level synthesis. IEEE Trans. on Computer-aided Design **15** (1996) 1354–1363
14. Kodukula, I., Pingali, K., Cox, R., Maydan, D.E.: An experimental evaluation of tiling and shackling for memory hierarchy management. In: Proceedings of the ACM International Conference on Supercomputing. (1999) 482–491
15. Deitz, S.J., Chamberlain, B.L., Snyder, L.: Eliminating redundancies in sum-of-product array computations. In: Proceedings of the ACM International Conference on Supercomputing. (2001) 65–77