Journal of Instruction-Level Parallelism 5 (2003) 1-28 Submitted 10/02; published 4/03

Exploiting Superword-Level Locality in Multimedia Extension Architectures

Jaewook Shin JAEWOOK@ISI.EDU
Jacqueline Chame JCHAME@ISI.EDU
Mary W. Hall MHALL@ISI.EDU
Information Sciences Institute, University of Southern California,
Suite 1001, 4676 Admiralty Way, Marina del Rey, CA 90292-6695

Abstract

In this paper, we describe an algorithm and implementation of locality optimizations for architectures with instruction sets such as Intel's SSE and Motorola's AltiVec that support operations on superwords, i.e., aggregate objects consisting of several machine words. We treat the large superword register file as a compiler-controlled cache, thus avoiding unnecessary memory accesses by exploiting reuse in superword registers. This research is distinguished from previous work on exploiting reuse in scalar registers because it considers not only temporal but also spatial reuse. As compared to optimizations to exploit reuse in cache, the compiler must also manage replacement, and thus, explicitly name registers in the generated code. We describe an implementation of our approach integrated with a compiler that exploits superword-level parallelism (SLP). We present a set of results derived automatically on 4 multimedia kernels and 2 scientific benchmarks. Our results show speedups ranging from 1.3 to 3.1X on the 6 programs as compared to using SLP alone, and we eliminate the majority of memory accesses.

1. Introduction

In response to the increasing importance of multimedia applications in embedded and general-purpose computing environments, many microprocessors now incorporate an expanded instruction set and architectural extensions specifically targeting multimedia requirements. The core component of such architectural extensions is a functional unit that can operate on aggregate objects, performing bit-level operations, or SIMD parallel operations on variable-sized fields in the object (*e.g.,* 8, 16, 32 or 64-bit fields). If the aggregate objects are larger than the size of a machine word, then they are called *superwords* [1]. Examples include Motorola's AltiVec and Intel's SSE, a descendant of MMX. If the same size as the machine word, then individual fields are referred to as *subwords* [2]. A related class of architectures employ processing-in-memory (PIM) technology to exploit the high memory bandwidth when processing logic is combined on chip with large amounts of DRAM; several PIM-based architectures rely on superword parallelism to make more effective use of available memory bandwidth [3, 4, 5, 6].

While multimedia extension and related architectures have been available for some time, convenient methodologies for developing application code that targets these extensions are in their infancy. There is recent compiler research for such architectures to automatically exploit *superword-level parallelism*, performing computations or memory accesses in parallel in a single instruction issue [1, 7, 8, 9, 10].

In this paper, we recognize an additional optimization opportunity not addressed by this previous work. An important feature of all such architectures is a register file of superwords (*e.g.,* each 128 bits wide in an AltiVec), usually in addition to the scalar register file. A set of 32 such superword registers represents a not insignificant amount of storage close to the processor. Accessing data from superword registers, versus a cache or main memory, has two advantages. The most obvious advantage is lower latency of accesses; even a hit in the L1 cache has at least a 1-cycle latency. Accesses to other caches in the hierarchy or to main memory carry much higher latencies. Another advantage is the elimination of memory access instructions, thus reducing the number of instructions to be issued.

In this paper, we treat the superword register file as a small compiler-controlled cache. We develop an algorithm and a set of optimizations to exploit reuse of data in superword registers to eliminate unnecessary memory accesses, which we call *superword-level locality*. We evaluate the effectiveness of these superword-level locality (SLL) optimizations through an implementation integrated with the algorithm for exploiting superword-level parallelism (SLP) presented in [1].

Our approach is distinguished from previous work on increasing reuse in cache [11, 12, 13, 14, 15, 16, 17, 18], in that the compiler must also manage replacement, and thus, explicitly name the registers in the code. As compared to previous work on exploiting reuse in scalar registers [18, 19, 20], the compiler considers not just temporal reuse, but also spatial reuse, for both individual statements and groups of references. Further, it also considers superword parallelism in making its optimization decisions. Exploiting spatial and group reuse in superword registers requires more complex analysis as compared to exploiting temporal reuse in scalar registers, to determine which accesses map into the same superword.

In conjunction with exploiting SLP, the algorithm performs what we call *superword replacement*, to replace accesses to contiguous array data with superword temporaries and exploit reuse by replacing accesses to the same superword with the same temporary. Following this code transformation, a separate compilation pass will be able to allocate superword registers corresponding to the superword temporaries. To enhance the effectiveness of superword replacement, it is combined with a loop transformation called *unroll-and-jam*, whereby outer loops in a loop nest are unrolled, and the resulting duplicate inner loop bodies are fused together. Unroll-and-jam reduces the distance between reuse of the same superword, when reuse is carried by an outer loop, and brings opportunities for superword replacement into the innermost loop body of the transformed loop nest. The optimization algorithm derives appropriate unroll factors for each loop in the nest that attempt to maximize reuse while not exceeding the number of available registers.

The contributions of this paper are as follows:

- An algorithm for exposing opportunities for compiler-controlled caching of data in superword register files using unroll-and-jam. The two main components of this algorithm are a model of the number of memory accesses and registers required associated with a set of unroll factors, and a strategy for navigating the search space of possible unroll factors.

- A description of a set of code transformations, which in aggregate we call superword replacement, for exploiting superword register reuse.

- Experimental results, derived automatically, comparing performance of six benchmarks/multimedia kernels optimized for parallelism only, SLP, and optimized for both parallelism and superword-level locality. Our results show speedups ranging from 1.3 to 3.1X as compared to using SLP alone, and we eliminate the majority of memory accesses.

This paper extends an earlier description of this work in several ways [21]. We have extended the algorithm and register requirements analysis to exploit group-temporal reuse across iterations of the transformed loop nest. We have also greatly expanded the description of code generation. In the experimental results description, we have improved the results and provided a more detailed breakdown of the contributions of the different techniques.

The remainder of the paper is organized into 8 sections. Section 2 motivates the problem and introduces terminology used in the remainder of the paper. Section 3 presents an overview of the superword-level locality algorithm. Section 4 describes how the algorithm computes the total number of registers required for exploiting reuse and the resulting number of memory accesses. Section 5 describes aspects of how the search space is navigated. Section 6 presents optimizations to actually achieve this reuse of data in superword registers. Section 7 presents experimental results derived automatically by an implementation in the Stanford SUIF compiler. Section 8 discusses related work and Section 9 presents conclusions and future work.

2. Background and Motivation

In many cases superword-level parallelism and superword-level locality are complementary optimization goals, since achieving SLP requires each operand to be a set of words packed into a superword, which happens, with no extra cost, when an array reference with spatial reuse is loaded from memory into a superword register. Therefore, in many cases the loop that carries the most superword-level parallelism also carries the most spatial reuse, and benefits from SLL optimizations. In this paper, we achieve SLL and SLP somewhat independently, by integrating a set of SLL optimizations into an existing SLP compiler [1]. The remainder of this section motivates the SLL optimizations.

Achieving locality in superword registers differs from locality optimization for scalar registers. To exploit temporal reuse of data in scalar registers, compilers use *scalar replacement* to replace array references by accesses to temporary scalar variables, so that a separate backend register allocator will exploit reuse in registers [19]. In addition, *unroll-and-jam* is used to shorten the distances between reuse of the same array location by unrolling outer loops that carry reuse and fusing the resulting inner loops together [19].

In contrast, a compiler can optimize for superword-level locality in superword registers through a combination of unroll-and-jam and *superword replacement*. These techniques not only exploit temporal reuse of data, but also spatial reuse of nearby elements in the same superword. In fact, even partial reuse of superwords can be exploited by merging the contents of two registers containing superwords that are consecutive in memory (see Section 6.4). Thus, as is common in multimedia applications [22], streaming computations with little or no temporal reuse can still benefit from spatial locality at the superword-register level, in addition to the cache level.

While cache optimizations are beyond the scope of this paper, we observe that the SLL optimizations presented here can be applied to code that has been optimized for caches using well-known optimizations such as unimodular transformations, loop tiling and data prefetching. When combining loop tiling for caches, superword-level parallelism and superword-level locality optimizations, the tile sizes should be large enough for superword-level parallelism, and for unroll-and-jam and superword replacement to be profitable.

These points are illustrated by way of a code example, with the original code shown in Figure 1(a). This example shows three optimization paths. Figure 1(d) optimizes the code to achieve

```
for(i=0; i<n; i++)
   for (j=0; j<n; j++)
        a[i][j] = a[i-1][j] * b[i] + b[i+1];
```

(a) Original loop nest.

```
for(i=0; i<n; i+=2)
   for (j=0; j<n; j++) {
        a[i][j] = a[i-1][j] * b[i] + b[i+1];
        a[i+1][j] = a[i][j] * b[i+1] + b[i+2];
   }
```

(b) Unroll-and-jam on the example in (a)(i loop).

```
tmp1 = b[0];
for(i=0; i<n; i+=2) {
   tmp2 = b[i+1];
   tmp3 = b[i+2];
   for (j=0; j<n; j++) {
        tmp4 = a[i-1][j] * tmp1 + tmp2;
        a[i+1][j] = tmp4 * tmp2 + tmp3;
        a[i][j] = tmp4;
   }
   tmp1 = tmp3;
}
```

(c) After scalar replacement on the code in (c).

```
for(i=0; i<n; i++)
   for (j=0; j<n; j+=sws)
        a[i][j:j+sws-1] = a[i-1][j:j+sws-1] * b[i] + b[i+1];
```

(d) After superword-level parallelism(j loop).

```
for(i=0; i<n; i+=2)
   for (j=0; j<n; j+= sws) {
        a[i][j:j+sws-1] = a[i-1][j:j+sws-1] * b[i] + b[i+1];
        a[i+1][j:j+sws-1] = a[i][j:j+sws-1] * b[i+1] + b[i+2];
   }
```

(e) Unroll-and-jam on the example in (d)(i loop).

```
tmp1[0:sws-1] = b[0:sws-1];
stmp1 = tmp1[0];
stmp2 = tmp1[1];
field = 2;
for(i=0; i<n; i+=2) {
   // 'field' denotes an index into 'tmp1' for stmp3
   if(field == 0)
        tmp1[0:sws-1] = b[i+2:i+sws+1];
   stmp3 = tmp1[field];
   for (j=0; j<n; j+= sws) {
        tmp2[0:sws-1] = a[i-1][j:j+sws-1] * stmp1 + stmp2;
        a[i+1][j:j+sws-1] = tmp2[0:sws-1] * stmp2 + stmp3;
        a[i][j:j+sws-1] = tmp2[0:sws-1];
   }
   stmp1 = stmp3;
   stmp2 = tmp1[field+1];
   field = (field+2)%sws;
}
```

(f) After superword replacement on code in (e)

Figure 1: Example code.

superword-level parallelism. Here, sws, an abbreviation for superword size, is the number of data elements that fit within a superword. For example, if a and b are 32-bit float variables, on a machine with 128-bit superwords, $sws = 4$. In Figures 1(b) and (c), we show how the original program can instead be optimized to exploit reuse in scalar registers, using unroll-and-jam and scalar replacement, respectively. In Figures 1(e) and (f), we combine these ideas, using unroll-and-jam and superword replacement, respectively, to transform the code in (d) for both superword-level parallelism and superword-level locality.

Table 1 shows how the three different optimization paths affect the number of array accesses to memory in the final code. The original code has n^2 reads and writes to array a and $2n^2$ reads to array b. Exploiting superword-level parallelism in loop j, as in Figure 1(d) reduces the number of reads and writes to array a by a factor of sws since each load or store operates on sws contiguous data items; for array b, there is no change since the array is indexed by i rather than j. If instead the code was optimized for scalar register reuse, as in Figure 1(c), we can reduce the number of array reads of a down by a factor of 2, and reads of b by a factor of n, with the number of writes remaining the

	Original Figure 1(a)	Scalar register reuse only Figure 1(c)	SLP only Figure 1(d)	SLP and SLL Figure 1(f)
Reads	$3n^2$	$n^2/2+n$	$2n^2+n^2/sws$	$(n^2/2+n)/sws$
Writes	n^2	n^2	n^2/sws	n^2/sws

Table 1: Number of array accesses under different optimization paths.

same. By combining superword-level parallelism and superword-level locality as in Figure 1(f), we see that the number of reads and writes is further reduced by a factor of sws. Figure 1(f) illustrates some of the challenges in exploiting reuse in superwords. Analysis must identify not just temporal, but also spatial reuse, and for both individual statements and groups of references. The compiler also must generate the appropriate code to exploit this reuse; for example, we select scalar fields of b from the superword, since we are not parallelizing the i loop.

The remainder of this paper describes how the compiler automatically generates code such as is shown in Figure 1(f), and the performance improvements that can be obtained with this approach.

3. Overview of Superword-Level Locality Algorithm

The superword-level locality algorithm has three main steps, as summarized below. Each step will be described in more detail in the three subsequent sections.

Step 1: Identifying Reuse. The first step of the algorithm is to identify both array references and loops carrying reuse. The array references carrying reuse are the ones for which superword replacement may be applicable. The loops carrying reuse are the ones to which the algorithm will consider applying unroll-and-jam.

Reuse between two distinct array references in an $n-$dimensional loop nest is determined from data dependences, in the form of *dependence vectors*, $d = \langle d_1, d_2, \ldots, d_n\rangle$[23]. A dependence vector captures the vector distance, in terms of the loop iteration space, such that the two references may map to the same memory location. Each vector element d_i may be either a constant integer, $+$ (a positive direction where the distance is not fixed), $-$ (a negative direction), or $*$ (the direction and distance are unknown). We refer to a dependence vector as being *lexicographically positive* if the first non-zero d_i is $+$ or a positive integer.

For the purposes of reuse, the relevant dependences carrying reuse are a subset, and are characterized as follows:

1. We consider only true dependences (writes followed by reads), input dependences (reads followed by reads), and output dependences (writes followed writes). Although output dependences do not capture reuse of the same data value, they suggest an opportunity for eliminating unnecessary writes back to memory. Anti-dependences (writes followed by reads) are not considered.

2. We consider only lexicographically positive dependences.

3. A dependence vector must be *consistent*, *i.e.*, the dependence distance in the iteration space must be constant, or it must be invariant with respect to one of the loops in the nest.

```
for(i=0; i<N; i+=4){
    vec1[0:3] = A[i:i+3];
    vec2[0:3] = A[i+8:i+11];
        ⋮
}
```

(a) Original

```
tmp[0:3] = A[i:i+3];
vec2[0:3] = A[i+4:i+7];
for(i=0; i<N; i+=4){
    vec1[0:3] = tmp[0:3];
    tmp[0:3] = vec2[0:3];
    vec2[0:3] = A[i+8:i+11];
        ⋮
}
```

(b) After exploiting reuse

Figure 2: Reuse Across Iterations

Applying unroll-and-jam to a loop i with a consistent dependence varying with respect to loop i can create loop-independent dependences in the innermost loop of the unrolled loop body. In the example in Figure 1(a), there is a true dependence between references $A[i][j]$ and $A[i-1][j]$ with distance vector $\langle 1, 0\rangle$. After unroll-and-jam, a loop-independent dependence is created between $A[i][j]$ in the first statement and $A[i][j]$ in the second statement of the loop body, creating a reuse opportunity.

In addition to reuse between copies of a reference created by unrolling, there can be reuse across loop iterations. References with consistent dependences carried by a loop have group reuse which can be exploited by using extra registers to hold the data across iterations. As in previous work [19], our algorithm exploits reuse across iterations of the innermost loop only, because exploiting reuse carried by an outer loop could potentially require too many registers to hold the data between uses. Figure 2 shows how reuse can be exploited across iterations of the innermost loop by using one register to keep the data that is reused on every two iterations.

For loop-invariant references, unroll-and-jam generates loop-independent dependences between the copies of the reference in the unrolled loop body, since the same location is being referenced by each copy.

Step 2: Determining unroll factors for candidate loops. The algorithm next determines the unroll factors for each candidate loop that carries reuse, as previously described, and for which unroll-and-jam is legal. The optimization goal is as follows.

> *Optimization Goal:* Find unroll factors $\langle X_1, X_2, ...X_n\rangle$ for loops 1 to n in an n-deep loop nest such that the number of memory accesses is minimized, subject to the constraint that the number of superword registers required does not exceed what is available.

The algorithm determines the unroll factors $\langle X_1, X_2, ...X_n\rangle$ by searching for the combination of unroll factors that satisfies the above optimization goal. To guide the search, the algorithm calculates the total number of registers required for exploiting reuse, which is the sum of the number of superwords accessed by the references in the loop body after unroll-and-jam is applied, plus the number of registers needed for holding data across iterations of the innermost loop. Section 4 describes how the algorithm computes the total number of registers required for exploiting reuse and the resulting number of memory accesses. Section 5 describes aspects of how the search space is navigated.

Step 3: Code Transformations - Unroll-and-Jam, Superword Replacement, and Related Optimizations. Once the unroll factors are decided, unroll-and-jam is applied to the loop nest. Array references are replaced with accesses to superword temporaries. As part of code generation, our compiler performs related optimizations to reduce the number of additional memory accesses and register requirements introduced by the SLP passes. These code transformations are the topic of Section 6.

4. Computing Registers Required and Memory Accesses

This section presents the computation of the number of registers required for exploiting data reuse in superword registers and the resulting number of memory accesses, which are the parameters used to guide the search for the combination of unroll amounts to be applied to the loop nest. The next subsection describes how the algorithm computes the *superword footprint*, which represents the number of superwords accessed by the unrolled iterations of the loop nest as a function of the unroll factors. Subsection 4.2 presents the computation of the extra registers needed for reusing data across loop iterations. The total number of registers and the corresponding number of memory accesses are computed in subsection 4.3.

4.1 Computing the Superword Footprint

This section presents the computation of the superword footprint of the references V in a loop nest, $F_L(V)$, after unroll-and-jam is applied to the nest with unroll factors $\langle X_1, X_2, ..., X_n\rangle$.

The algorithm for computing the superword footprint for a loop nest first partitions the references in the loop into groups of *uniformly generated references* [18], that is, references to the same array such that, for each array dimension, the array subscripts differ only by a constant term[1]. Then, for each group of references, it computes the number of superwords accessed in the unrolled loop body. Finally, the total number of superwords is computed as the sum of those of each group of uniformly generated references.

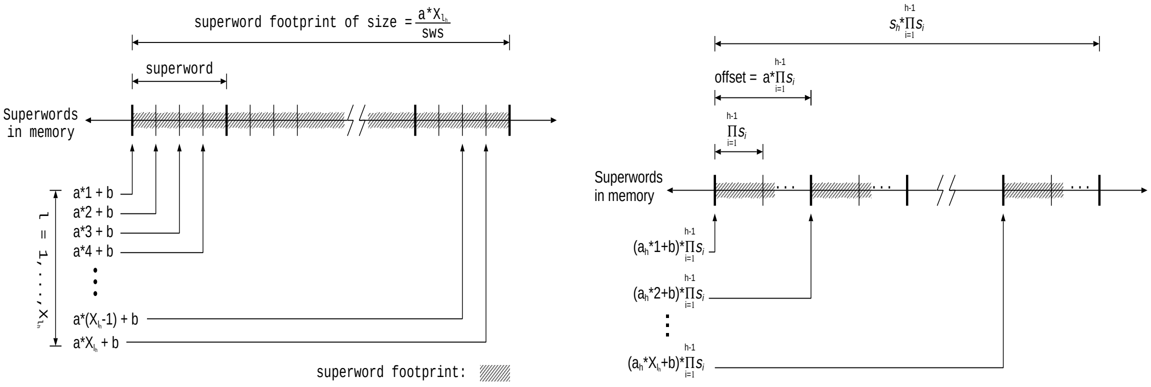
We first discuss how to compute the superword footprint of a single reference as a function of the unroll factors of each unrolled loop. Then we discuss how to compute the superword footprint of a group of uniformly generated references. The superword footprint of a group may be smaller than the sum of the individual fooptrints, since the same superword may be accessed by two or more copies of the original references when the loops are unrolled.

Our method determines the number of superword registers required to hold the data accessed by the loop references in the unrolled loops. However, extra registers may be needed to, for example, align a superword operand which is already kept in superword registers. That is, the computation may require more registers than those needed for storing the data. Therefore, we reserve some scratch registers for manipulating data and compute the number of registers needed just for storing the data accessed in the unrolled loops.

To simplify the presentation, we assume a loop nest of depth n where all array references have array subscripts that are affine functions of a single index variable (SIV subscripts)[2]. We also assume that each p-dimensional array referenced by the loop is defined as $A[s_p][s_{p-1}]\dots[s_1]$, where s_h is

[1] We assume that two or more references that access the same array, but are not uniformly generated, access distinct data in memory, which results in a conservative estimate of the number of superwords accessed by the group and of the number of registers required.

[2] Our current implementation can handle affine SIV subscripts and certain affine MIV subscripts.


superword footprint of size = $\frac{a*X_{l_h}}{sws}$
superword
Superwords in memory
a*1 + b
a*2 + b
a*3 + b
a*4 + b
a*(X_{l_h}-1) + b
a*X_{l_h} + b
l = 1, ..., X_{l_h}
superword footprint: ▨
$s_h * \prod_{i=1}^{h-1} s_i$
offset = $a * \prod_{i=1}^{h-1} s_i$
$\prod_{i=1}^{h-1} s_i$
Superwords in memory
$(a_h*1+b)*\prod_{i=1}^{h-1} s_i$
$(a_h*2+b)*\prod_{i=1}^{h-1} s_i$
$(a_h*X_{l_h}+b)*\prod_{i=1}^{h-1} s_i$

(a) $h = 1$ and $a_h < sws$

(b) $h \neq 1$

Figure 3: Superword footprint of a single reference.

the size of dimension h, $1 \leq h \leq p$. Dimension 1 is the lowest dimension of the array, *i.e.,* the dimension in which consecutive elements are in consecutive memory locations. A reference v to array A is then of the form $A[a_p * l_p + b_p][a_{p-1} * l_{p-1} + b_{p-1}] \ldots [a_1 * l_1 + b_1]$. Thus, a reference with SIV subscripts has each array dimension h associated with just a single loop index variable in the nest, and the loop index variable associated with h is represented as l_h. We also assume that the arrays are aligned to a superword in memory and that the loops are normalized.

4.1.1 SUPERWORD FOOTPRINT OF A SINGLE REFERENCE

For each reference v with array subscripts $a_h * l_h + b$, where h is the array dimension and l_h is the loop index variable appearing in subscript h, the number of superwords accessed by all copies of v when l_h is unrolled by X_{l_h} is given by the *superword footprint* of v in l_h, or $F_{l_h}(v)$.

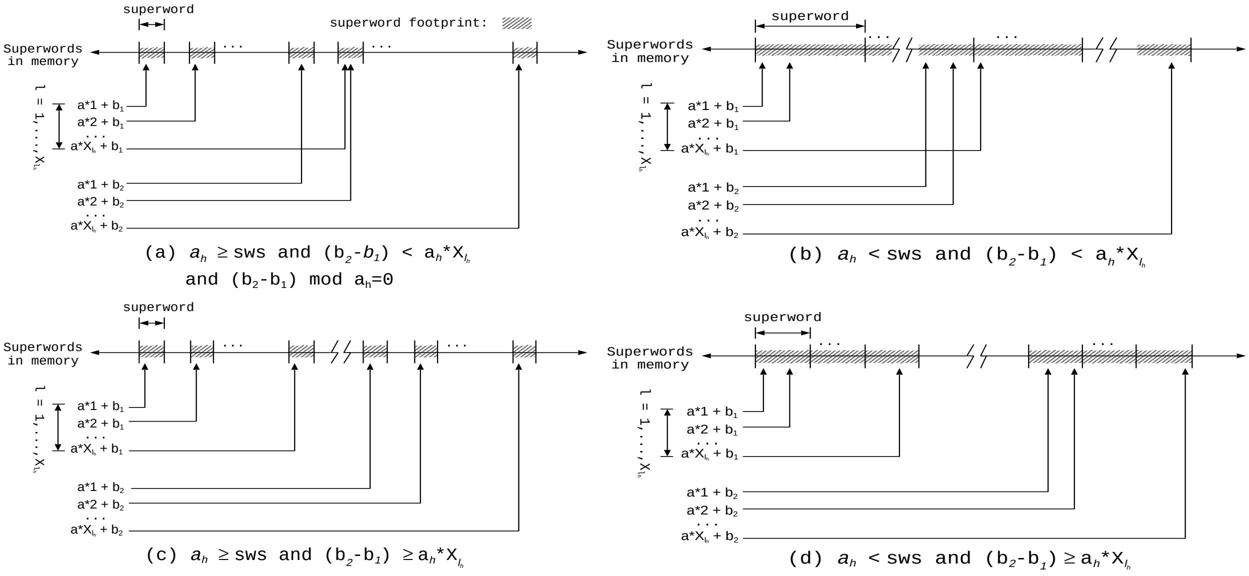
When dimension h is the lowest array dimension ($h = 1$), the superword footprint is given by Equation (1). Equation (1a) corresponds to the footprint of a loop-invariant reference. Equation (1b) corresponds to the footprint of a reference with self-spatial reuse within a superword, as illustrated in Figure 3(a), and (1c) holds when the reference has no spatial reuse.

$$F_{l_h}(v) = \begin{cases} 1 & \text{(a) if } a_h = 0 \\ \left\lceil \frac{X_{l_h} * a_h}{sws} \right\rceil & \text{(b) if } a_h < sws \\ X_{l_h} & \text{(c) if } a_h \geq sws \end{cases} \tag{1}$$

When h is one of the higher dimensions, $1 < h \leq p$, and loop l_h is unrolled, the offset between the footprints of each copy of v is $a_h * \prod_{i=1}^{h-1} s_i$, where s_i is the size of the i^{th} array dimension, as shown in Figure 3(b). Assuming that the size of the lowest array dimension (s_1) is larger than sws, which is usually the case in practice for realistic array dimensions, each copy of v in the unrolled loop body corresponds to a separate footprint, as shown in Figure 3(b). Therefore the size of the footprint of v in l_h is the sum of the X_{l_h} disjoint footprints, and is recursively defined by Equation (2), where $F_{l_1}(v)$ is computed as in Equation (1).

$$\begin{aligned} F_{l_h}(v) &= X_{l_h} * F_{l_{h-1}}(v) \\ &= (\prod_{i=2}^{h} X_{l_i}) * F_{l_1}(v) \end{aligned} \tag{2}$$

$$F_{l_h}(v_1, v_2) = \begin{cases} X_{l_h} + (b_2 - b_1)/a_h & \text{(a) if } a_h \geq sws \text{ and } (b_2 - b_1) < a_h * X_{l_h} \text{ and} \\ & \quad (b_2 - b_1) \bmod a_h = 0 \\ \lceil (a_h * X_{l_h} + b_2 - b_1)/sws \rceil & \text{(b) if } a_h < sws \text{ and } (b_2 - b_1) < a_h * X_{l_h} \\ F_{l_h}(v_1) + F_{l_h}(v_2) & \text{(c) otherwise} \end{cases} \tag{3}$$



(a) $a_h \geq$ sws and (b_2-b_1) < a_h*X_{l_h} and (b_2-b_1) mod a_h=0

(b) a_h < sws and (b_2-b_1) < a_h*X_{l_h}

(c) $a_h \geq$ sws and (b_2-b_1) $\geq a_h$*X_{l_h}

(d) a_h < sws and (b_2-b_1) $\geq a_h$*X_{l_h}

Figure 4: Superword footprint of a group of references.

For a single reference, the number of superword registers required to keep the superword footprint given by Equation (1) and the number of scalar registers that would be required if the same unroll factors were used differ only when $a_h < sws$, that is, when spatial reuse can be exploited in superword registers. For a group of uniformly generated references the analysis must also consider group reuse, as discussed next.

4.1.2 SUPERWORD FOOTPRINT OF A GROUP OF REFERENCES

The number of superwords accessed by a group of uniformly generated references $V = \{v_1, v_2, ..., v_m\}$ when loop l_h is unrolled by X_{l_h} is the superword footprint of the group, $F_{l_h}(V)$. The superword footprint of a group consists of the union of the footprints of the individual references, as some of the reference footprints may overlap, depending on the distance between the constant terms in the array subscripts.

The footprints of two uniformly generated references may overlap in dimension h only if they overlap in all dimensions higher than h. For example, the footprints of references $A[2i][j+2]$ and $[2i+1][j]$ do not overlap in the highest (row) dimension, since the first reference accesses the even-numbered rows of the array and the second accesses the odd-numbered rows. Therefore the footprints cannot overlap in the lowest (column) dimension. On the other hand, the footprints of $A[2i][j+2]$ and $A[2i+4][j]$ overlap in the row dimension for iterations i_1, i_2, $1 \leq i_1, i_2 \leq X_i$, such that $2i_1 = 2i_2 + 4$. For the iterations of i in which the footprints overlap in the row dimension, the footprints may overlap in the column dimension if there exist iterations j_1, j_2, $1 \leq j_1, j_2 \leq X_j$, such that $j_1 + 2 = j_2$.

The superword footprint $F_L(V)$ of a group V, following unroll-and-jam, is computed as follows. First, the array dimensions with array subscripts that are a function of any of the unrolled loops are identified. Then, for each such dimension h, from highest to lowest dimension, the footprint is computed assuming that the footprints of the references in the group overlap in the higher dimensions. For each dimension $h > 1$, the algorithm partitions references into subsets such that each subset corresponds to a disjoint footprint in dimension h. Then, for each subset, the algorithm recursively computes the footprint in dimension $h - 1$, as we now describe.

Dimension h is the lowest dimension ($h = 1$). We first compute the group footprint of two array references, and then we extend it for m references. The footprint of group $V = \{v_1, v_2\}$, where references v_1 and v_2 have lowest dimension subscripts $a_h * l_h + b_1$ and $a_h * l_h + b_2$ such that $b_1 \leq b_2$, when loop l_h is unrolled by X_{l_h} is given by Equation (3) in Figure 4. Equations (3a) and (3b) apply when the two footprints overlap, that is, when $(b_2 - b_1) < a_h * X_{l_h}$, as shown in Figures 4(a) and (b). When the footprints do not overlap, the group footprint is the sum of the individual footprints, as in Equation (3c), with examples in Figures 4(c) and (d).

In Figure 4(a), the references have no self-spatial reuse, that is, $a_h \geq sws$, and each individual footprint is a set of X_{l_h} superwords. The footprints overlap if $(b_2 - b_1)$ is evenly divided by a_h and there exists an integer value k, $1 \leq k \leq X_{l_h}$, such that $k = 1 + (b_2 - b_1)/a_h$. This case corresponds to Equation (3a), which computes the group footprint precisely when the two references have group-temporal reuse. In Figure 4(b), both references have self-spatial reuse within a superword, that is, $a_h < sws$. The corresponding footprint size is given by Equation (3b). In Figure 4(c), v_1 has no self-spatial reuse and each copy of v_1 in the unrolled loop body accesses a distinct superword, and the same is true for v_2. In Figure 4(d) both v_1 and v_2 have self-spatial reuse.

The footprint of a group $V = \{v_1, v_2, ..., v_m\}$, with array subscripts $a_1 * l_1 + b_i$ such that $1 \leq i \leq m$ and $b_1 \leq b_2 \leq ... \leq b_m$, is computed by first partitioning V into subgroups with disjoint footprints in the lowest dimension, as follows. A subgroup $V_i = \{v_{i_{min}}, v_{i_{min+1}}, ..., v_{i_{max}}\}$ is defined by lowest dimension subscripts $a_1 * l_1 + b_j$, where $\forall j, \quad i_{min} < j \leq i_{max}$,

$$\begin{aligned}
&(b_{j-1} \leq b_j) \wedge \\
&(b_j - b_{j-1} < a_1 * X_{l_1}) \wedge \\
&(b_{i_{min}} = b_1 \vee b_{i_{min}} - b_{i_{min}-1} \geq a_1 * X_{l_1}) \wedge \\
&(b_{i_{max}} = b_m \vee b_{i_{max}+1} - b_{i_{max}} \geq a_1 * X_{l_1})
\end{aligned} \tag{4}$$

Then the group footprint V is computed as the sum of the disjoint footprints of sets V_i, as in (5).

$$F_{l_h}(V) \quad = \quad \sum_i F_{l_h}(V_i) \tag{5}$$

The footprint of each subgroup V_i is computed by extending Equation (3) to $m > 2$ references. For example, when the references in V have self-spatial reuse, as in Equation (3b) ($a_1 < sws$), each subgroup V_i has a footprint consisting of contiguous superwords, since $b_j - b_{j-1} < a_1 * X_{l_1}$ for all j such that $i_{min} < j \leq i_{max}$. The footprint of V_i consists of the union of the individual footprints, with size given by Equation (6).

$$\begin{aligned} F_{l_h}(V_i) &= F_{l_h}(\{v_{i_{min}}, ..., v_{i_{max}}\}) \\ &= \left\lceil \frac{a_1 * X_{l_1} + b_{i_{max}} - b_{i_{min}}}{sws} \right\rceil \end{aligned} \quad (6)$$

For example, if $sws = 4$ and $X = 4$, group $V = \{A[i], A[i+2], A[i+5], A[i+12], A[i+14]\}$ can be partitioned into two subgroups $V_1 = \{A[i], A[i+2], A[i+5]\}$ and $V_2 = \{A[i+12], A[i+14]\}$ with disjoint superword footprints. Since the references have self-spatial reuse, each individual footprint and the footprint of each subgroup is a set of contiguous superwords. The total number of superwords accessed by the references in V is the sum of the disjoint footprints of sets V_1 and V_2, as in (7).

$$F_{l_1}(V) = F_{l_1}(V_1) + F_{l_1}(V_2) = \left\lceil \frac{1*4+5-0}{4} \right\rceil + \left\lceil \frac{1*4+14-12}{4} \right\rceil = 5 \quad (7)$$

Dimension h is not the lowest dimension ($h \neq 1$). When h is one of the higher dimensions, the superword footprint of $V = \{v_1, v_2, ..., v_m\}$ in loop l_h is again the union of the individual footprints.

From Section 4.1.1, the footprint of each reference v_i in the unrolled loop body consists of a set of X_{l_h} disjoint footprints (each footprint corresponding to a copy of v_i created by unrolling), and the offset between each pair of consecutive footprints is $a_h * \prod_{i=1}^{h-1} s_i$, where s_i is the size of dimension i.

Therefore the footprints of different references in the group may overlap, depending on the values of a_h, b_j and the unroll factor X_{l_h}. The footprints of two uniformly generated references v_1 and v_2 overlap in dimension h if there exists an integer value k, $1 \leq k \leq X_{l_h}$ that satisfies Condition (8):

$$a_h * k + b_1 = a_h + b_2. \quad (8)$$

that is, if $(b_2 - b_1)\% a_h = 0$ and $(b_2 - b_1)/a_h + 1 \leq X_{l_h}$. Furthermore, if there exists k satisfying the above condition, the footprints of the last $X_{l_h} - k + 1$ copies of v_1 in the unrolled loop body overlap with those of the first $X_{l_h} - k + 1$ copies of v_2. The footprint of $\{v_1, v_2\}$ is then given by Equation (9).

$$F_{l_h}(v_1, v_2) = (k-1) * F_{l_{h-1}}(v_1) + (X_{l_h} - k + 1) * F_{l_{h-1}}(v_1, v_2) + (k-1) * F_{l_{h-1}}(v_2) \quad (9)$$

To compute the size of the entire footprint of V in l_h, our algorithm partitions V into subsets $V_i = \{v_{i_{min}}, ..., v_{i_{max}}\}$ such that, for any j, $i_{min} < j \leq i_{max}$, the pair $\{v_{j-1}, v_j\}$ satisfies Condition (8). The footprint of V_i is the union of the footprints of its reference set and is computed by extending Equation (9) to more than two references.

4.2 Registers for Reuse Across Iterations

In addition to superword registers for exploiting reuse in the body of the transformed loop nest, extra superword registers may be required for exploiting reuse across iterations of the innermost loop for references with group-temporal reuse carried by the innermost loop n of the transformed loop nest.

To compute the number of registers needed to exploit group-temporal reuse across iterations of loop n, the algorithm examines groups of references that have consistent dependences carried by n.[3] Assume that unroll-and-jam has been applied to outer loops in a nest. After subsequently unrolling the innermost loop, extra registers are required if the reuse distance between references prior to unrolling loop n is larger than the unroll amount, *i.e.*, if $d_n > X_n$, as in Figure 2, where $d_n = 8$ and $X_n = 4$.

Let $C = \{v_1, v_2, ...v_m\}$ be a set of references that is a subset of a uniformly generated set, and, prior to unrolling the innermost loop resulting from unroll-and-jam by X_n, each pair $\langle v_i, v_{i+1}\rangle$ in C has a consistent dependence $d^i = \langle 0, 0, ..., d_n^i\rangle$, $d_n^i > 0$. Also, assume that the array subscript of the lowest dimension of each reference v_i in C is of the form $a_i * n + b_i$, and that $b_1 \leq b_2 \leq ... \leq b_m$. Unrolling loop n generates X_n copies of each original reference v_i in the body of the transformed loop nest.

When d_n^i is a multiple of the unroll factor X_n, each pair of copies of references $\langle v_i,\, v_{i+1}\rangle$ will reuse data after $\frac{d_n^i}{X_n}$ iterations. When d_n^i is not a multiple of X_n, some copies of a reference will reuse data after $\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1$ iterations of n, while others will have a reuse distance of $\left\lceil \frac{d_n^i}{X_n} \right\rceil$ requiring one more register per copy. Thus, each pair of copies of references $\langle v_i,\, v_{i+1}\rangle$ requires at most $\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1$ additional superword registers to keep the data across iterations of the innermost loop.

The number of registers required to exploit reuse across iterations of n by all pairs of copies is the number of registers required for each pair times the number of registers required to keep the superword footprint of reference v_i in the transformed loop nest:

$$R_A(v_i, v_{i+1}) \quad = \quad (\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1) \times F_L(v_i) \tag{10}$$

Equation (10) may overestimate the number of registers if the footprint component ($F_L(v_i)$) overestimates registers, or for certain copies of references if d_n^i is not a multiple of X_n.

The total number of registers required for exploiting reuse across iterations for set C with leading reference v_1 is given by:

$$R_A(C) \quad = \quad \sum_{1 \leq i < m} \left((\left\lceil \frac{d_n^i}{X_n} \right\rceil - 1) \times F_L(v_1) \right) \tag{11}$$

4.3 Putting It All Together

Subsections 4.1 and 4.2 describe the computation of the number of registers required to exploit reuse in the body of the innermost loop (superword footprint) and across iterations of the innermost loop, assuming that unroll-and-jam has been applied the loop nest. This section presents the computation of the total number of registers required and the total number of memory accesses in the innermost loop of the transformed loop nest, which are the metrics used to prune and guide the search for unroll factors described in Section 3.

The total number of registers required to exploit reuse is the sum of the superword footprint of the references in the innermost loop of the transformed loop nest and the number of registers needed for exploiting reuse across iterations of the same innermost loop.

The superword footprint of the references, $F_L(V)$, is computed as in subsection 4.1. The total number of extra registers required for exploiting reuse across iterations of the innermost loop is

[3] Note that such references, if their lowest dimension varies with n, may also have group-spatial reuse across loop iterations. However, our algorithm focuses on exploiting group-temporal reuse across iterations, since most of the group-spatial reuse is achieved within the body of the unrolled loop.

computed as in subsection 4.2, for each set C of loop-variant references with consistent dependences carried by the innermost loop.

The total number of superword registers required is then:

$$R(V) \;=\; F_L(V) + \sum_C R_A(C) \tag{12}$$

The total number of memory accesses in the innermost loop of the transformed loop nest is the sum of the memory accesses of each group C of references that are variant with the innermost loop n and have consistent dependences carried by n. For each group C, the number of memory accesses is given by the superword footprint of the leading reference of the group, v_1^c:

$$M(C) = F_L(v_1^c) \tag{13}$$

The total number of memory accesses is then:

$$M(V) \;=\; \sum_c F_L(v_1^c) \tag{14}$$

5. Search Algorithm

As previously stated, the goal of the search algorithm is to identify the unroll factors for the loops in the loop nest such that the number of memory accesses is minimized, without exceeding available registers. Thus, we must consider an $n-$dimensional search space, where each dimension has the number of elements corresponding to the iteration count of the loop. A full global search of this search space is prohibitively expensive, especially for deep loop nests or large loop bounds. Thus, we use a number of strategies for pruning the search space.

First, we eliminate from the search loops that do not carry reuse or for which unroll-and-jam is not safe. Further, we rely on the observation that the number of registers required monotonically increases with the unroll factor of a loop, assuming that all other unroll factors are fixed. Thus, we need not search beyond the unroll factors that exceed available registers. This latter point significantly prunes the search space in that the number of registers is usually fairly small (*e.g.,* 32 superword registers on the AltiVec), so that the search is concentrated on fairly small unroll factors. These pruning strategies are used in our current implementation, and at least for the programs in this study, are quite effective at making the search practical.

Further pruning is possible by making the additional observation that for each unrolled loop l, the amount of reuse of an array reference with reuse carried by l increases with the unroll factor X_l. Therefore reuse, like the register requirement calculation, is a monotonic, non-decreasing function of the unroll factor for each loop, given that the unroll factor of all other loops is fixed. Thus, within each dimension, holding all other unroll factors constant, binary search can be used rather than searching all points. We can also increase unroll factors by amounts corresponding to the superword size without much loss of precision, rather than considering each possible unroll factor, since the register requirements increase stepwise as a function of superword size. Additional pruning techniques that take into account the hardware's capability to take advantage of the results of optimization have been used in prior work [19, 24].

Our implementation navigates the search space from innermost loop to outermost loop, for the applicable loops in the nest, varying the unroll factor of one loop while keeping the unroll factors of all other loops fixed. Within a dimension of the search space, the lowest number of memory accesses will be derived at the largest unroll factor that meets the register constraint. However, lower unroll factors may also have the same estimate of memory accesses (because reuse is monotonically non-decreasing), so we identify the lowest unroll factor with the equivalent estimate of memory accesses. Then, the implementation considers the next applicable outer loop and the applicable inner loops nested inside it, and in a particular dimension, each time it reaches the largest unroll factor that meets the register constraint, it compares the estimated number of memory accesses to the lowest estimate so far to determine if a better solution has been found. The final result of the algorithm is the unroll factors corresponding to the best solution.

As a subtle point, when unroll-and-jam is applied from outermost to innermost loop, unrolling the inner loop does not affect data access patterns or reuse distance. For this reason, inner loop unrolling is not performed in earlier work [19]. In our context, however, because of the relationship between superword-level parallelism and superword replacement, inner loop unrolling exposes opportunities for superword loads and stores and thus can impact the analysis of register requirements. Nevertheless, when reuse is exploited across iterations of the innermost loop body as described in Section 4.2, it is not necessary to unroll the innermost loop beyond the superword size to achieve the goal of considering register requirements in conjunction with superword-level parallelism. Note, however, that smaller unroll factors for the innermost loop may be selected, if an unroll-and-jam of an outer loop carries more parallelism and reuse.

Although this search should theoretically find the optimal solution, according to our optimization criteria, in fact the solution is not guaranteed to result in the fewest number of memory accesses, for a number of reasons. First, in a few cases as noted, the register requirement analysis defined in the previous section must conservatively approximate. Second, it is difficult to estimate the register requirements used to hold temporaries, so we conservatively approximate this as well. Third, there is a tradeoff between using extra registers to hold values across iterations, as discussed in Section 4.2, versus using them to actually exploit reuse within the transformed innermost loop body. In fact, in general the algorithm does not take into consideration the amount of reuse resulting from performing superword replacement on specific references; replacing some references has more impact on decreasing memory accesses than others.

This section and the previous one have described how the compiler analyzes the code to identify reuse, register requirements and the unroll factors leading towards the lowest number of memory accesses. In the next section, we describe how these analyses are used in transforming the code to achieve the desired result.

6. Code Generation

In the previous section, we showed how consideration of superwords instead of scalar variables greatly increases the complexity of determining the number of registers and memory accesses associated with exploiting reuse under different unroll amounts. In this section, we further discuss the increased complexity of code generation when performing superword replacement instead of scalar replacement. The chief source of code generation complexity is the need for superword objects to be properly *aligned*, as in the following examples.

When performing memory operations, the architecture may actually require that an access be aligned at superword boundaries. For example, the AltiVec ignores the last four bits of an address when performing a superword load or store. In such an architecture, when an access is not aligned at a superword boundary, the compiler or programmer must read/write two adjacent superwords. A series of additional instructions *packs* the two superwords for reads or *unpacks* a superword into its corresponding two superwords for writes. Even on architectures that support memory accesses not aligned at superword boundaries, such as Intel's SSE, there is a performance penalty on unaligned accesses because the hardware must perform this realignment.

To perform an arithmetic or logical operation on two superword registers, the fields of the two operands must also be aligned. For example, to add the third and fourth fields of one superword register to the first and second fields of another, one of the registers must be shifted by two fields. Consider also the following example:

```
for i = 1, n
   c[i] = a[2i] + b[i]
```

The access to a has a stride of 2, while the access to b has a unit stride. Thus, the compiler or programmer must first pack the even elements of a into a superword register before adding them to the elements of b. A third example occurs when exploiting partial reuse of a superword where data in a register must be aligned to accommodate the next operation.

In the SLP compiler, the default solution to alignment involves packing data through memory. The SLP compiler allocates superword variables by declaring them using a special vector type designation, which is interpreted by the backend compiler to align the beginning of the variable to a superword boundary in memory. The start of each dimension of an array of such objects should also be aligned, by padding if necessary. Under these assumptions, the SLP compiler can detect when operations are unaligned. Unaligned data is packed into an aligned superword in memory before being loaded into a superword register, and is unpacked before storing back to memory.[4]

In summary, alignment is a key consideration in code generation, and the overhead of performing alignment operations can be quite high. Further, alignment operations may require a number of additional superword registers, and in some cases, may result in additional accesses to memory not accounted for by the model in the previous section. In this section, we show how to achieve the number of registers derived by our model through a set of code transformations, presented in the order in which they are performed by our compiler. In addition to superword replacement, described in Section 6.2, we also describe how index set splitting is used to align accesses to the beginning of an iteration in Section 6.1, and how our compiler eliminates additional memory accesses resulting from packing through memory for alignment in Section 6.3. We illustrate how these transformations collaborate with each other by way of an example in Figure 5, which is a simplified FIR filter.

6.1 Index Set Splitting

A simple way to reduce the need for alignment operations, when applicable, is to perform index set splitting on loops. For example, in Figure 5(b), the initial access to out[1] refers to the

[4] For architectures that support copying between scalar and superword register files, such as Intel's SSE and DIVA, this packing can be performed more efficiently through register copies.

```
1)   for (i = 1; i < 64; i++)
2)        out[i] = 0.0;
3)
4)   for (i = 256; i < 320; i++)
5)        for (j = 0; j < 256; j++)
6)             out[i-256] = out[i-256] + in[i-j] * coe[j];
```

(a) Original

```
1)   for (i = 1; i < 4; i++){
2)        out[i] = 0.0;
3)   }
4)   for (i = 4; i < 64; i++){
5)        out[i] = 0.0;
6)   }
7)   for (i = 256; i < 320; i++){
8)        for (j = 0; j < 256; j++){
9)             out[i - 256] = out[i - 256] + in[i - j] * coe[j];
10)       }
11)  }
```

(b) After index set splitting

```
1)   for (i = 1; i < 4; i++){
2)        out[i] = 0.0;
3)   }
4)   for (i = 4; i < 64; i += 4){
5)        out[i + 0] = 0.0;
6)        out[i + 1] = 0.0;
7)        out[i + 2] = 0.0;
8)        out[i + 3] = 0.0;
9)   }
10)  for (i = 256; i < 320; i += 8){
11)       for (j = 0; j < 256; j += 8){
12)            out[i + 0 - 256] = out[i + 0 - 256] + in[i + 0 - (j + 0)] * coe[j + 0];
13)            out[i + 0 - 256] = out[i + 0 - 256] + in[i + 0 - (j + 1)] * coe[j + 1];
14)                                 ⋮
15)            out[i + 7 - 256] = out[i + 7 - 256] + in[i + 7 - (j + 7)] * coe[j + 7];
16)       }
17)  }
```

(c) After unroll-and-jam

Figure 5: Code Generation Example

second field of a superword, assuming `out[0]` is aligned at a superword boundary. Through index set splitting, the portion of the loop from line 4-6 will always perform aligned accesses. This transformation is always safe, and is profitable whenever it increases the number of aligned memory accesses.

We assume index set splitting is performed prior to the SLP compiler. The loop is transformed so that accesses corresponding to a particular reference in the main loop body are aligned to superword boundaries. If there are multiple references and different choices for index set splitting are needed to align specific references, we select a representative reference that, if aligned through index set splitting, will also maximize alignment for other references. The reference selected must have unit stride within the innermost loop.

Let i be the loop index variable for the innermost loop, and lb and ub are the lower and upper bounds for i. To derive the loop bounds for the copies of the innermost loop resulting from index set splitting, we begin with the starting address, $addr$, of the reference when $i = lb$, where $addr = base + \textit{offset}$. Here, $base$ refers to the beginning of the lowest dimension of the selected array, and

```
                  ⋮
1)   flat1 = *((float *)&vec0 + 3);
2)   flat2 = *((float *)&vec1 + 0);
3)   flat3 = *((float *)&vec1 + 1);
4)   flat4 = *((float *)&vec1 + 2);
5)   *((float *)&vec2 + 0) = flat1;
6)   *((float *)&vec2 + 1) = flat2;
7)   *((float *)&vec2 + 2) = flat3;
8)   *((float *)&vec2 + 3) = flat4;
9)   vec4 = vec_add(vec3, vec2);
10)  vec_st(vec4, i * 4 + 0, (float *)&out[-63]);
11)  vec5 = vec_ld(i * 4, (float *)&out[-63]);
12)  flat5 = *((float *)&vec6 + 2);
13)  flat6 = *((float *)&vec7 + 2);
14)  *((float *)&vec8 + 0) = flat5;
15)  *((float *)&vec8 + 1) = flat6;
                  ⋮
```

(d) After SLP compilation

```
                  ⋮
1)   flat1 = *((float *)&vec0 + 3);
2)   flat2 = *((float *)&vec1 + 0);
3)   flat3 = *((float *)&vec1 + 1);
4)   flat4 = *((float *)&vec1 + 2);
5)   *((float *)&vec2 + 0) = flat1;
6)   *((float *)&vec2 + 1) = flat2;
7)   *((float *)&vec2 + 2) = flat3;
8)   *((float *)&vec2 + 3) = flat4;
9)   vec4 = vec_add(vec3, vec2);
10)  flat5 = *((float *)&vec6 + 2);
11)  flat6 = *((float *)&vec7 + 2);
12)  *((float *)&vec8 + 0) = flat5;
13)  *((float *)&vec8 + 1) = flat6;
                  ⋮
```

(e) After superword replacement

```
                  ⋮
1)   temp1 = replicate(vec0, 3);
2)   temp2 = replicate(vec1, 0);
3)   temp3 = replicate(vec1, 1);
4)   temp4 = replicate(vec1, 2);
5)   vec2 = shift_and_load(temp1, temp1, 4);
6)   vec2 = shift_and_load(vec2, temp2, 4);
7)   vec2 = shift_and_load(vec2, temp3, 4);
8)   vec2 = shift_and_load(vec2, temp4, 4);
9)   vec4 = vec_add(vec3, vec2);
10)  temp1 = replicate(vec6, 2);
12)  temp2 = replicate(vec7, 2);
11)  vec8 = shift_and_load(temp1, temp1, 4);
13)  vec8 = shift_and_load(vec8, temp2, 12);
                  ⋮
```

(f) After packing in registers

Figure 5: Code Generation Example(Continued)

offset is the offset within that dimension. (Recall that the beginning of each dimension is aligned at superword boundaries.)

The lower bound (*split*) of the main loop body is computed by the following equation.

$$split = \begin{cases} lb & \text{if } \textit{offset} \bmod \textit{sws} = 0 \\ lb + \textit{sws} - (\textit{offset} \bmod \textit{sws}) & \text{if } \textit{offset} \bmod \textit{sws} \neq 0 \end{cases} \tag{15}$$

If lb is constant, $split$ can be computed at compile time. Otherwise, it is computed at run time. In the example in Figure 5, *offset* for `out[1]` is 1, so if *sws* $= 4$, then $split = 4$.

6.2 Superword Replacement

Superword replacement removes redundant loads and stores of superword variables, using superword temporaries instead. We assume that this code transformation will be followed by register allocation that places these variables in registers. For example, in Figure 5(d) and (e), the store and load at statements 10 and 11 can both be eliminated, and `vec4` can be used in place of `vec5` in subsequent statements. Superword replacement is also affected by alignment, in that we detect redundant loads and stores by identifying distinct memory operations that refer to the same aligned superword, even if the addresses are not identical.

The compiler recognizes opportunities for superword replacement by determining that addresses and offsets for different memory accesses fit within the same superword, and verifies that there are no intervening kills to the memory locations. The current implementation uses *value numbering* [25] to detect such opportunities. Value numbering is a well-known compiler technique for detecting redundant computation, but it is sensitive to operand and operator ordering. To increase the success of value numbering, we first preprocess the code so that memory access operations are rewritten into a canonical form, constant folding has been applied to simplify addresses, and alignment is taken into account. As earlier stated, all memory accesses are aligned at superword boundaries, so if an unaligned address appears in a memory access, the resulting access will be aligned to the preceding superword boundary. The preprocessing performs this alignment in software so that redundant accesses will be identified by value numbering.

The current implementation of superword replacement is more restrictive than what was presented in Section 3. Value numbering operates on a basic block at a time so we cannot exploit reuse across iterations of the unrolled loop body. This is because we are performing this transformation after the SLP compiler has flattened the loop structure to gotos and labels. The dependence information used to perform the register requirement analysis cannot easily be reconstructed from such low-level code. In an implementation where SLP and SLL are more tightly integrated, it should be possible to perform superword replacement as a byproduct of the analysis in Section 3.

6.3 Packing in Superword Registers

As previously described, packing in memory is performed to align superword objects. Memory packing moves data elements from a set of locations in memory (*sources*) to a superword location (*destination*) so that the destination superword contains contiguous data, aligned to a superword boundary or to another operand. For example, in Figure 5(e), superword variables `vec0` and `vec1` are the sources and superword variable `vec2` is the destination for memory packing in lines 1-8.

Our implementation performs a transformation we call *register packing* to optimize memory packing operations. A series of memory loads and stores for scalar variables are replaced by superword operations on registers, as shown in Figure 5 (f). We identify a destination as a superword data type that is the target of a series of scalar store instructions into its fields, such as `vec2` in the example. The corresponding sources are identified by finding preceding loads of these scalar variables. If the inputs to these loads are fields of superword data types, then these superwords are the sources. In the example, `flat1` is stored into a field of `vec2`, and there is a preceding load of `flat1` that copies a field of source `vec0`. Once we find such a pattern, we verify the safety of this tranformation by guaranteeing that there are no intervening modifications or uses of either the scalar variables or destination superwords between loading the scalar variables and completion of storing into the destination. We also verify that the destination statements ultimately produce con-

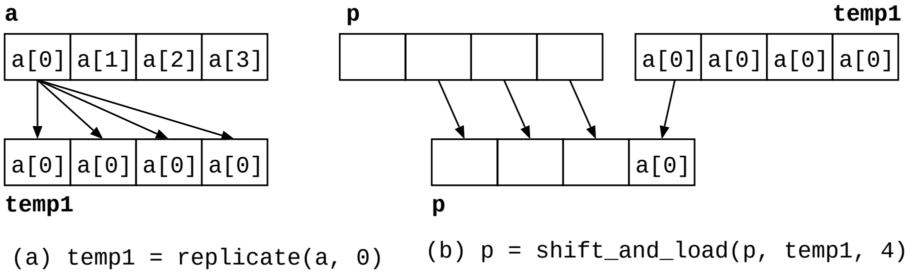


Figure 6: Operations used for packing in registers

tiguous data in the superword. We define *source* and *destination indices* as the fields in the source and destination superword variables, respectively. For example, the source index of `vec0` is 3 in line 1 of the example.

Once the compiler identifies sources and destinations, it transforms the code to replace memory accesses with operations on superword registers. The register packing transformation takes advantage of two instructions that are common in multimedia extension architectures. *Replicate* replicates one element of a source register to all elements of a temporary output register (Figure 6(a)). *Shift-and-load* takes two input registers. The first input register is a temporary, and is shifted left by the number of bytes specified by the third argument. The same number of fields is taken from the second input register, which is a temporary derived from a source superword, to fill the output temporary register (Figure 6(b)). Simply stated, we are shifting each source element into the destination superword, in order, so that the final result is a destination superword that corresponds to contiguous aligned data.

The steps of the register packing transformation are as follows.

1. We sort the destination statements in increasing order of their destination indices. We then sort the source statements to correspond to the ordering of the destination statements, so that, for example, the scalar variable associated with the first source statement is the same as the scalar variable associated with the first destination statement.

2. For each source statement, in sorted order, we generate a replicate statement whose two inputs are the source superword and the source index, and the output is a superword temporary. For example, as in Figure 5(f), we have replaced line 1 of Figure 5(e) with $temp1 = replicate(vec0, 3)$.

3. We replace each destination statement, in sorted order, with a `shift_and_load` operation. The first input is the destination superword. The second input is the temporary generated by the `replicate` of the corresponding source statement. The third argument, the shift amount, usually involves shifting by a single superword field. For the last destination field, the shift amount is the difference, in bytes, between the *sws* and the last destination field. For completely filled destination superwords, it will also be just a single field. For example, in lines 1-8 of Figure 5(e), the destination superword is completely filled, so the shift amount is always a single 4-byte field. In lines 10-13, however, only the first two fields are filled, so the shift amount of the last destination statement is a total of 12 bytes

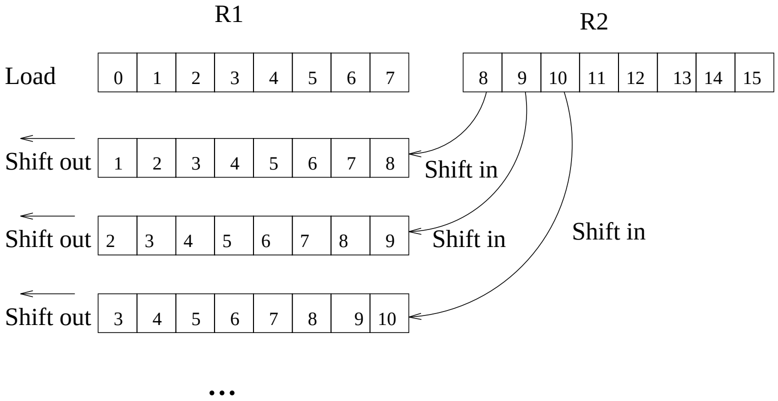


Figure 7: Shifting

4. Source statements are deleted if the scalar variables are not live beyond the corresponding destination statements.

6.4 An Example: Shifting for Partial Reuse

Spatial reuse within a superword happens when distinct loop iterations access different data in the same superword. *Partial spatial reuse* of superwords occurs when distinct loop iterations access data in consecutive superwords in memory, partially reusing the data in one or both superwords, as shown by the example in Figure 5 (a), and illustrated graphically in Figure 7. In this example, as before assuming that $sws = 4$, array reference $in[i-j]$ has partial spatial reuse in loop i. For a fixed value of i and j, the data accessed in iteration $\langle i, j \rangle$ consists of the last three words of the superword accessed in iteration $\langle i-1, j \rangle$, plus the first word of the next superword in memory. This type of reuse can be exploited by shifting the first word out of the superword, and shifting in the next word, as in Figure 7. As partially shown in Figure 5(c) and (f), only four superwords need to be loaded for the data accessed in the 64 copies of $in[i-j]$ in the loop body, after shifting is applied. Before shifting, $in[i-j]$ had to be loaded from memory (and possibly aligned) for each of the four copies of $in[i-j]$ in the loop body.

This shifting opportunity arises frequently in both signal and image processing applications, where one object is compared to a subcomponent of another object, such as the example in Figure 5(a). We detect these opportunities through the analysis described in Section 3. The optimization shown in Figure 7 falls out from the combination of unroll-and-jam, alignment operations generated by the SLP compiler, superword replacement and register packing.

7. Experimental Results

This section presents an experiment that demonstrates the dramatic performance improvements that can be derived from compiler-controlled caching in superword registers. We describe an implementation that incorporates superword register locality optimizations into an existing compiler exploiting superword-level parallelism [1]. We present a set of results on four multimedia kernels and two scientific applications, derived automatically from our implementation.

7.1 Implementation and Methodology

Figure 8 illustrates the system we have developed for this experiment, which uses the Stanford SUIF compiler as its underlying infrastructure [26]. The input to the system is a C program, which is then optimized by passes in SUIF, including our Superword Locality analysis described in Section 3, followed by the Superword-Level Parallelism (SLP) optimization passes by Larsen and Amarasinghe[1], and finally, an optimization pass that performs superword replacement as described in Section 6.2 to steer the compiler to obtain the reuse in superword registers that the SLL algorithm determined was possible.

This ordering of passes was selected primarily for implementation convenience, since we were building on the existing SLP compiler implementation. The SLP passes operate on the code at a low level, where it is difficult to reconstruct the loop structure and array access expressions. Thus, register requirement analysis and unroll-and-jam were applied prior to SLP, rather than afterward, as was suggested by the examples in Section 2. Superword replacement must follow SLP, which is the reason the components of our algorithm are performed on either side of SLP. Note that both the SLP passes and SLL employ loop unrolling, but for different reasons. The SLP compiler operates on basic blocks and unrolls the innermost loop of a loop nest to convert loop-level parallelism into basic-block parallelism. The SLL algorithm performs unroll-and-jam to expose locality in basic blocks. However, the loop that carries the most spatial locality at the superword level is often the loop that carries the most superword-level parallelism. Therefore, it is a reasonable choice to use the SLL algorithm to expose both parallelism and locality in the loop body while suppressing the unrolling originally performed by the SLP compiler.

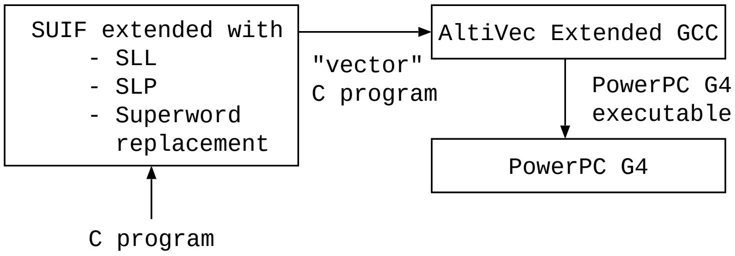


Figure 8: Implementation.

The output from the SUIF portion of the system is an optimized C program, augmented with special superword data types and operations. Currently, the resulting code is passed to a Gnu C backend, modified to support superword data types and operations for the PowerPC AltiVec instruction-set architecture extensions. Each superword operation corresponds, in most cases, to a single instruction in the AltiVec ISA. The role of the GCC backend includes replacing the vector operations with the corresponding AltiVec superword instruction, and allocating the vector data types to the superword registers. The resulting code is executed on a 533 MHz Macintosh PowerPC G4, which has a superword register file consisting of 32 128-bit registers.

7.2 Performance Measurements

We have applied the previously-described implementation to four of the five multimedia kernels and the two scientific programs from the Specfp95 benchmark suite for which execution time speedups were reported in Larsen and Amarasinghe, summarized in Table 2 [1]. As a first step, we verified

Name	Description	Data Width	Input Size
VMM	Vector-matrix multiply	32-bit float	512 elements
FIR	Finite impulse response filter	32-bit float	256 filter, 1M signal
YUV	RGB to YUV conversion	16-bit integer	32K elements
MMM	Matrix-matrix multiply	32-bit float	512 elements
swim	Shallow water model	32-bit float	Specfp95 reference input
tomcatv	Mesh generation	32-bit float	Specfp95 reference input

Table 2: Benchmark programs.

that we could reproduce their previously reported results. For purposes of comparison, we initially followed the same methodology established in Larsen and Amarasinghe [1]: (1) we used the same programs; (2) all versions of the code were compiled on the AltiVec without optimization; and, (3) baseline measurements were derived by compiling the unparallelized code for the PowerPC G4. We are using an updated implementation of SLP from what was published, as well as a faster target machine and new releases of GCC and the Linux operating system, so there are some differences in results, but they are very minor.

Larsen and Amarasinghe were unable to use optimization on the AltiVec-extended GCC backend at the time of their study, but in the intervening time, this Motorola-supplied backend has become more robust. For the results presented in this section, we modify the methodology to perform "-O3" optimizations. To understand the overall benefits of exploiting compiler-controlled caching in superword registers, we have compared the results of the full system with those obtained when SLP is used alone. For this reason, we report results where SLP is applied to the original codes and compare these results to the full system.

We show three sets of results. First, in Table 3, we show the number of vector, scalar and total memory accesses for the baseline and the full system. Our approach eliminates from 38% to 69% of the vector loads and stores in the four kernels, and over 85% in SWIM and TOMCATV. We also eliminate over 90% of the scalar loads and stores in the four kernels, and over 35% in SWIM and TOMCATV using register packing, as described in Section 6.3. When combined, more than 50% of memory accesses are eliminated.

Figure 9 shows how these reductions in instructions translates into speedups over SLP. To isolate the benefits of individual components of our system, we measure the performance of the code at several stages of the optimization process. The first bar, normalized to 1, shows the results of SLP alone. The second bar, called Unrolled+SLP, shows the results of running the first portion of the SLL algorithm, described in Section 3, which performs unroll-and-jam on the loop nest to expose opportunities for superword reuse, and following up with SLP. This bar isolates the impact of unrolling, since it is not until after SLP that this reuse is actually exploited. Also, because it is reordering the iteration space to bring reuse closer together, this version will also obtain locality benefits in the data cache. Thus, this bar provides the cache locality benefits of unroll-and-jam, which can be compared against the additional improvements from superword register locality. The third bar, Superword Replacement, provides speedup using superword replacement, as described in Section 6.2. The final bar, entitled Register Packing, shows the additional improvement due to this technique, described in Section 6.3.

Name	Mem. Acc	SLP only(baseline)	SLP+SLL+RegPack	Removed(%)
VMM	Scalar	301,989,888	0	100.00
	Vector	100,663,297	50,462,723	49.87
	Total	402,653,185	50,462,723	87.47
FIR	Scalar	1,113,940,672	82,031,104	92.64
	Vector	196,558,849	120,631,297	38.63
	Total	1,310,499,521	202,662,401	84.54
YUV	Scalar	9,400	0	100.00
	Vector	52,428,801	23,756,801	54.69
	Total	52,438,201	23,756,801	54.70
MMM	Scalar	135,267,328	525,312	99.61
	Vector	167,772,161	50,397,187	69.96
	Total	303,039,489	50,922,499	83.20
swim	Scalar	17,150,342,657	8,920,336,007	47.99
	Vector	8,495,723,139	1,200,754,698	85.87
	Total	25,646,065,796	10,121,090,705	60.54
tomcatv	Scalar	599,038,032	384,070,586	35.89
	Vector	284,631,621	9,915,592	96.51
	Total	883,669,653	393,986,178	55.41

Table 3: The number of dynamic memory accesses.

Overall, we see that in combination, applications achieve speedups between 1.3 and 3.1 over SLP alone, with an average of 2.2X. Consideration of TOMCATV and SWIM shows that both programs have little temporal reuse, although there is a small amount of spatial reuse that is exploited with our approach, particularly in TOMCATV. We are obtaining a locality benefit due to unroll-and-jam. We also observe additional SLP due to index set splitting, motivated by the need to create a steady-state loop where the data is aligned to a superword boundary. The four other programs show a significant improvement from superword replacement. For VMM, MMM and FIR, there are also huge gains due to register packing.

In Figure 10, we further explore the relationship between superword replacement and register packing. The first bar, which is normalized to 1, shows the Unrolled+SLP version (the second bar in the previous figure). The second bar is the Unrolled+SLP+SWR result from the previous figure, but this time it is normalized to Unrolled+SLP. To show the isolated benefit of register packing without superword replacement, we applied register packing to the Unroll+SLP version, obtaining the results shown in the third bar (Unroll+SLP+RP) of Figure 10. The final bar is the result of applying all of the optimizations. As might be expected from the previous figure, register packing, either in isolation or in conjunction with superword replacement, does not impact the results for YUV, swim or tomcatv. We see that for VMM and MMM, register packing yields about the same improvement when applied prior to superword replacement than afterward. Especially interesting are the results for FIR, because the speedup is much larger when superword replacement and register packing are applied together than when they are applied separately. On further investigation, we found that the Unroll+SLP+RP version suffered from register spilling. Superword replacement removes the majority of the superword variables used in the Unroll+SLP+RP version, which in turn reduces register pressure. This result is consistent with the goal of the algorithm in Section 3. We selected unroll factors based on the assumption that superword replacement would be performed.

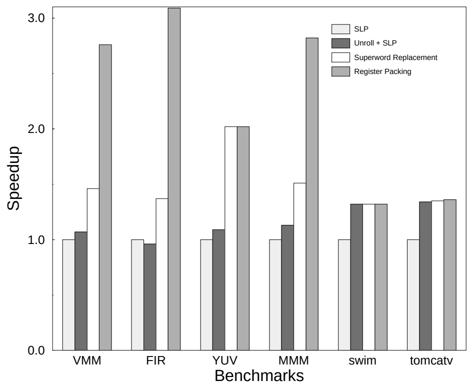


Figure 9: Speedups over SLP alone.

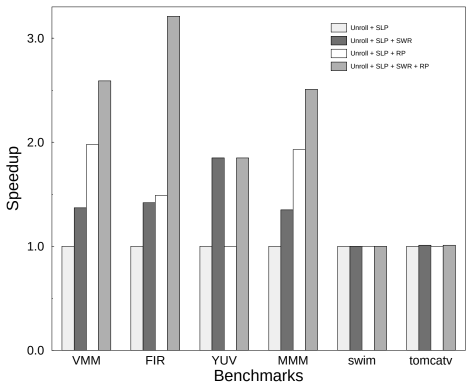


Figure 10: Impact of register packing.

Without superword replacement, there is register pressure after unrolling, and this is magnified by register packing because it introduces additional superword variables.

In summary, the SLL techniques presented in this paper dramatically reduce the number of memory accesses and yield significant performance improvements across these 6 programs. Thus, this paper has demonstrated the value of exploiting locality in superword registers in architectures that support superword-level parallelism such as the AltiVec.

8. Related Research

For well over a decade, a significant body of research has been devoted to code transformations to improve cache locality, most of it targeting loop nests with regular data access patterns [27, 28, 29, 30]. Loop optimizations for improving data locality, such as tiling, interchanging and skewing, focus on reducing cache capacity misses. Of particular relevance to this paper are approaches to tiling for cache to exploit temporal and spatial reuse; the bulk of this work examines how to select tile sizes that eliminate both capacity misses and conflict misses, tuned to the problem and cache sizes [31, 11, 12, 13, 14, 15, 16, 17, 18, 32]. The key difference between our work and that of tiling for caches is that interference is not an issue in registers. Therefore, models that consider conflict misses are not appropriate. Further, our code generation strategy must explicitly manage reuse in registers.

There has been much less attention paid to tiling and other code transformations to exploit reuse in registers, where conflict misses do not occur, but registers must be explicitly named and managed. A few approaches examine mapping array variables to scalar registers [18, 33, 20]. Most closely related to ours is the work by Carr and Kennedy, which uses scalar replacement and unroll-and-jam to exploit scalar register reuse [19]. Like our approach, in deriving the unroll factors, they use a model to count the number of registers required for a potential unrolling to avoid register pressure, and they replace array accesses, which would result in memory accesses, with accesses to temporaries that will be put in registers by the backend compiler. Their search for an unroll factor is constrained by register pressure and another metric called *balance* that matches memory access time to floating point computation time. Our approach is distinguished from all these others in that the model for register requirements must take spatial locality into account, we replace array accesses with superwords rather than scalars, and we also consider the optimizations in light of superword parallelism.

There are several recent compilation systems developed for superword-level parallelism [1, 7, 8, 9, 10]. Most, including also commercial compilers [34, 35], are based on vectorization technology [7, 9]. In contrast, Larsen and Amarasinghe devised a superword-level parallelization system for multimedia extensions [1]. They point out that there are many differences between the multimedia extension architectures and vector architectures, such as short vectors, ease of mixing with scalar instructions, and need for alignment of memory accesses [36]. They argue that their algorithm for finding superword-level parallelism from a basic block instead of a loop nest is much more effective than using vectorization-based techniques. None of the above approaches exploit reuse in the superword register file.

9. Conclusion

This paper presents an algorithm for compiler-controlled caching in superword register files. The algorithm is applicable to multimedia extensions such as Intel's SSE, PowerPC's AltiVec, and also to Processor-in-memory (PIM) architectures with support for superword operations.

We implemented our approach in an existing compiler targeting superword-level parallelism. We presented experimental results, derived automatically, comparing the performance of six benchmarks/multimedia kernels optimized for parallelism only, using SLP, and optimized for both parallelism and locality. Our results show speedups ranging from 1.3 to 3.1X, and an average of 2.2X, on the 6 programs as compared to using SLP alone, and most memory accesses are removed.

The approach taken here that separates optimizations for SLL and SLP is convenient for implementation purposes, since we are building upon the work of others. Further, as there are now a few other compilers that exploit superword-level parallelism [7, 8, 9, 10], the same can be used to extend these existing systems to incorporate compiler-controlled caching in superword registers. Ideally, however, an optimizer that integrates the superword parallelism and locality techniques could be even more effective. For example, in a combined algorithm, selection of which loops to parallelize could also take superword-level locality into account. A combined algorithm is the subject of future work.

10. Acknowledgments

The authors wish to thank Samuel Larsen and Saman Amarasinghe for providing their SLP implementation. We wish to especially thank Samuel Larsen for his tremendous support. We also wish to thank all the students in our research group for providing the underlying infrastructure for this work, including Yoon-Ju Lee, Byoungro So and Rommel Dongre.

References

[1] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Conference on Programming Language Design and Implementation*, (Vancouver, BC Canada), pp. 145–156, June 2000.

[2] R. Lee, "Subword parallelism with max2," *IEEE Micro*, vol. 16, pp. 51–59, Aug. 1996.

[3] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang, "Evaluation of existing architectures in IRAM systems." In First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, June 1997.

[4] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *ACM International Conference on Supercomputing*, Nov. 1999.

[5] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, and T. Sterling, "Microservers: A new memory semantics for massively parallel computing," in *ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[6] D. Elliott, M. Snelgrove, and M. Stumm, "Computational RAM: a memory-SIMD hybrid and its application to DSP," in *IEEE 1992 Custom Integrated Circuit Conference*, pp. 30.6.1 – 30.6.4, 1992.

[7] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, 2000.

[8] G. Cheong and M. S. Lam, "An optimizer for multimedia instruction sets," in *The Second SUIF Compiler Workshop*, (Stanford University, USA), Aug. 1997.

[9] D. J. DeVries, "A vectorizing suif compiler: Implementation and performance," Master's thesis, University of Toronto, 1997.

[10] K. Asanovic and J. Beck, "T0 engineering data." UC Berkeley CS technical report UCB/CSD-97-930.

[11] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *The SIGPLAN '95 Conference on Programming Language Design and Implementation*, (La Jolla, CA), June 1995.

[12] K. Esseghir, "Improving data locality for caches," Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[13] C. Fricker, O. Temam, and W. Jalby, "Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply," *TOPLAS*, vol. 17, pp. 561–575, July 1995.

[14] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: An analytical representation of cache misses," in *Proceedings of the 1997 ACM International Conference on Supercomputing*, (Vienna, Austria), July 1997.

[15] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, California), pp. 228–239, October 1998.

[16] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimization of blocked algorithms," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 63–74, 1991.

[17] O. Temam, E. Granston, and W. Jalby, “To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts,” in *ACM International Conference on Supercomputing*, (Portland, OR), Nov. 1993.

[18] M. E. Wolf, *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.

[19] S. Carr and K. Kennedy, “Improving the ratio of memory operations to floating-point operations in loops,” *ACM Transactions on Programming Languages and Systems*, vol. 15(3), pp. 400–462, July 1994.

[20] A. F. M. Jimenez, J.M. Llaberia and E. Morancho, “Index set splitting to exploit data locality at the register level,” Tech. Rep. UPC-DAC-1996-49, Universitat politecnica de Catalunya, 1996.

[21] J. Shin, J. Chame, and M. W. Hall, “Compiler-controlled caching in superword register files for multimedia extension,” in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, (Charlottesville, Virginia), September 2002.

[22] P. Ranganathan, S. Adve, and N. Jouppi, “Performance of image and video processing with general-purpose processors and media ISA extensions,” in *International Symposium on Computer Architecture*, May 1999.

[23] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[24] B. So, M. W. Hall, and P. C. Diniz, “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*, (Berlin, Germany), June 2002.

[25] S. S. Muchnick, *Advanced Compiler Design and Implementation*. 340 Pine St. Sixth Floor, San Francisco, CA 94104-3205, USA: Morgan Kaufmann, 1997.

[26] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, “Maximizing multiprocessor performance with the SUIF compiler,” *IEEE Computer*, vol. 29, pp. 84–89, Dec. 1996.

[27] J. Ferrante, V. Sarkar, and W. Thrash, “On estimating and enhancing cache effectiveness,” in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, (Santa Clara, California), pp. 328–343, August 1991.

[28] S. Carr, K. S. McKinley, and C.-W. Tseng, “Compiler optimizations for improving data locality,” in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, California), pp. 252–262, October 1994.

[29] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” in *Proceedings of the SIGPLAN ’91 Conference on Programming Language Design and Implementation*, (Toronto), pp. 30–44, June 1991.

[30] M. J. Wolfe, "More iteration space tiling," in *Proceedings of Supercomputing '89*, (Reno, Nevada), pp. 655–664, November 1989.

[31] J. Chame and S. Moon, "A tile selection algorithm for data locality and cache interference," in *International Conference on Supercomputing*, pp. 492–499, 1999.

[32] G. Rivera and C. Tseng, "A comparison of compiler tiling algorithms," in *the 8th International Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands*, Mar. 1999.

[33] S. Carr and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software—Practice and Experience*, vol. 24, no. 1, pp. 51–77, 1994.

[34] Veridian, *VAST/AltiVec Features*, June 2001. http://www.psrv.com/altivec_feat.html.

[35] Metrowerks, *CodeWarrior version 7.0 data sheet*, 2001. http://www.metrowerks.com/pdf/mac7.pdf.

[36] S. Larsen, E. Witchel, and S. Amarasinghe, "Increasing and detecting memory address congruence," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, (Charlottesville, Virginia), September 2002.