Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures

Jaewook Shin, Jacqueline Chame and Mary W. Hall
Information Sciences Institute
University of Southern California
{jaewook,jchame,mhall}@isi.edu

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 2.8X 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* [20]. 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* [22]. 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 [2, 17, 3, 11].

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 [20, 27, 8, 10, 1].

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 [20].

Our approach is distinguished from previous work on increasing reuse in cache [9, 12, 14, 15, 16, 19, 28, 30], in that

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

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

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 [30, 5, 23], 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.

The contributions of this paper are as follows:

- An algorithm for exposing opportunities for compiler-controlled caching of data in superword register files.
- A description of a set of optimizations, 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 2.8X as compared to using SLP alone, and we eliminate the majority of memory accesses.

The remainder of the paper is organized into 5 sections. Section 2 motivates the problem and introduces terminology used in the remainder of the paper. Section 3 presents the main superword-level locality algorithm, which performs a set of transformations and an optimization search that exposes opportunities for reuse of data in superword registers. Section 4 presents optimizations to actually achieve this reuse of data in superword registers. Section 5 presents experimental results derived automatically by an implementation in the Stanford SUIF compiler. Section 6 discusses related word and Section 7 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 [20]. 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 [5]. 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 [5]. In conventional architectures with scalar register files, spatial locality can only be obtained in caches.

In contrast, a compiler can optimize for superword-level locality in superword registers locality 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 4.3). Thus, as is common in multimedia applications [25], streaming computations with little or no temporal reuse can still benefit from spatial locality at the superword-register level, as well as at 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(b) optimizes the code to achieve superword-level parallelism. Here, sws, an abbreviation for superword size, is the number of data ele-

```
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++)
   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];
```

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

```
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];
   }
```

(c) 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;
}
```

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

```
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 (b)(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.

ments 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(c) and (d), 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 (b) 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(b) 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(d), 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 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 Superword-Level Locality Algorithm

The superword-level locality algorithm has four main steps, as summarized in the next subsection. At the heart of the algorithm is an approach for counting both memory accesses and register requirements for storing reused data, which is the subject of the subsequent subsection.

3.1 Steps of Algorithm

Step 1: Identifying Reuse. First, we identify array variables and loops carrying temporal or spatial reuse. We examine the dependence graph, looking for references that have loop-carried consistent dependences (*i.e.,* constant dependence distances) or are loop invariant with one of the loops, and so have opportunities for data reuse that can be exposed by unroll-and-jam.

Applying unroll-and-jam to a loop with a loop-variant reference creates loop-independent dependences in the un-

rolled 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, creating a reuse opportunity. Similarly, spatial and group-temporal reuse can be exposed by unroll-and-jam when a reference has a loop-carried dependence with the loop that traverses the lowest array dimension. For loop-invariant references, unroll-and-jam generates loop-independent dependences between the copies of the reference in the unrolled loop body.

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

> *Optimization Goal:* Find unroll factors $\langle X_1, X_2, ...X_n\rangle$ for loops 1 to n in a 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 search algorithm uses the reuse information and the number of registers available to prune the search space, as follows. Loops that carry no reuse are not included in the search. Next, we observe 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 is a monotonic, non-decreasing function of the unroll factor for each loop, given that the unroll factor of all other loops are fixed. The algorithm uses this property to prune the search space, avoiding searching for all possible unroll factors for a given loop. It traverses the search space by varying the unroll factor of one loop while keeping the unroll factor of all other loops fixed. A binary search within a dimension can further prune the search. Also, the unroll factor of each loop, given that all other unroll factors are fixed, is limited by the number of registers available. Once the search finds an unroll factor for a given loop that exceeds the register limit, it prunes all larger unroll factors for that loop from the search space.

To guide the search towards the above optimization goal, we calculate the *superword footprint*, which represents the number of superwords accessed by the unrolled iterations of the loop nest, as a function of the unroll factor. The superword footprint can be used both to count how many registers are required to hold the accessed data, as well as how many memory accesses remain in the loop nest. Assuming that all variables are kept in registers when the superword footprint fits in the superword register file, the number of memory accesses associated with a set of references is simply the superword footprint for the references multiplied by the bounds of the loops in which they are nested after unrolling. Our method for selecting unroll factors based on required superword registers differs from related approaches oriented towards scalar registers [5], accounting for not only temporal but also spatial and group reuse. In the next subsection, we describe in detail the calculation of the superword footprint.

Step 3: Unroll-and-Jam and Superword Replacement. Once the unroll factors are decided, the loop nest is transformed and array references are replaced with accesses to superword temporaries, as discussed in Section 4.

3.2 Computing the Superword Footprint

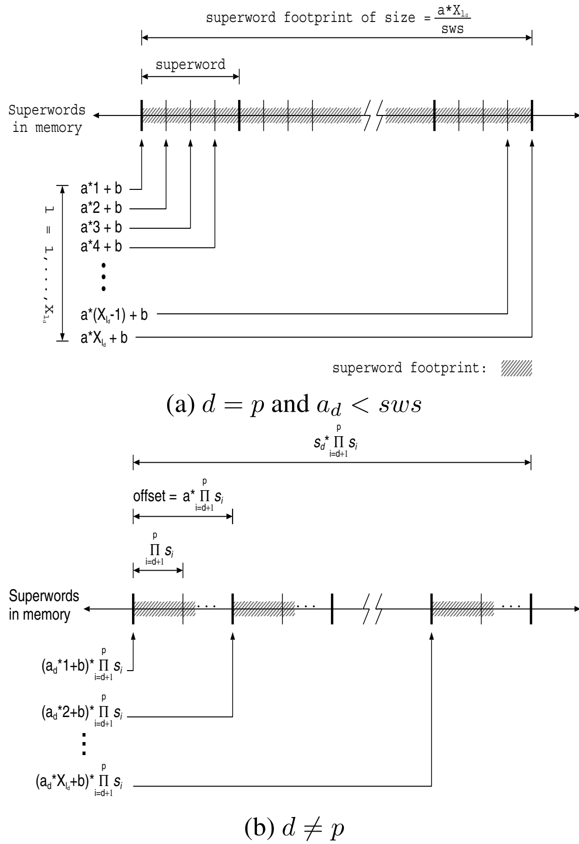
The algorithm for computing the superword footprint for a loop nest first partitions the references in the loop into groups of *uniformly generated references* [30], that is, references to the same array such that, for each array dimension, the array subscripts differ only by a constant term[1]. Then, for each group of references, it computes the registers needed to keep the data accessed in the unrolled loop body. Finally, the total number of registers is computed as the sum of those of each group of uniformly generated references. We first discuss how to compute the registers required for a single reference as a function of the unroll factors of each unrolled loop. Then we discuss how to compute the register requirements for a group of uniformly generated references. The registers required for such a group may be smaller than the sum of the registers required for each reference, if computed individually, 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 $A[s_1][s_2]\ldots[s_p]$, where s_d is the size of dimension d, $1 \leq d \leq p$. Dimension p is the lowest dimension of the array, *i.e.,* the dimension

[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 registers.

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



(a) $d = p$ and $a_d < sws$

(b) $d \neq p$

Figure 2. Superword footprint of a single reference.

in which consecutive elements are in consecutive memory locations. A reference v to array A is then of the form, $A[a_1 * l_1 + b_1][a_2 * l_2 + b_2] \dots [a_p * l_p + b_p]$. Similarly, the array subscripts of the uniformly generated references $v_1, v_2, \dots v_m$ in dimension d are $a_d * l_d + b_1$, $a_d * l_d + b_2$, ..., $a_d * l_d + b_m$, respectively. Thus, a reference with SIV subscripts has each array dimension associated with just a single loop index variable in the nest. We also assume that the arrays are aligned to a superword in memory and that the loops are normalized.

3.2.1 Superword Footprint of a Single Reference

For each reference v with array subscripts $a_d * l_d + b$, where d is the array dimension and l_d is the loop variable appearing in subscript d, the number of registers required to keep the data referenced by v when l_d is unrolled by X_{l_d} is given by the *superword footprint* of v in l_d, or $F_{l_d}(v)$. The superword footprint consists of the superwords accessed by all copies of v resulting from unrolling.

When dimension d is the lowest array dimension ($d = p$), 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 2(a), and (1c) holds when the reference has no spatial reuse.

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

When d is one of the higher dimensions, $1 \leq d < p$, and loop l_d is unrolled, the offset between the footprints of each copy of v is $a_d * \prod_{i=d+1}^{p} s_i$, where s_i is the size of the ith array dimension, as shown in Figure 2(b). Assuming that the size of the lowest array dimension (s_p) 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 2(b). Therefore the size of the footprint of v in l_d is the sum of the X_{l_d} disjoint footprints, and is recursively defined by Equation (2), where $F_{l_p}(v)$ is computed as in Equation (1).

$$\begin{aligned} F_{l_d}(v) &= X_{l_d} * F_{l_{d+1}}(v) \\ &= (\prod_{i=d}^{p-1} X_{l_i}) * F_{l_p}(v) \end{aligned} \quad (2)$$

For a single reference, the number of superword registers given by Equation (1) and the number of scalar registers that would be required if the same unroll factors were used differ only when $a_d < sws$, 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.

3.2.2 Superword Footprint of a Reference Group

The number of registers required to keep a group of uniformly generated references $V = \{v_1, v_2, ..., v_m\}$ when loop l_d is unrolled by X_{l_d} is the superword footprint of the group, $F_{l_d}(V)$. 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 d only if they overlap in all dimensions higher than d. 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

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

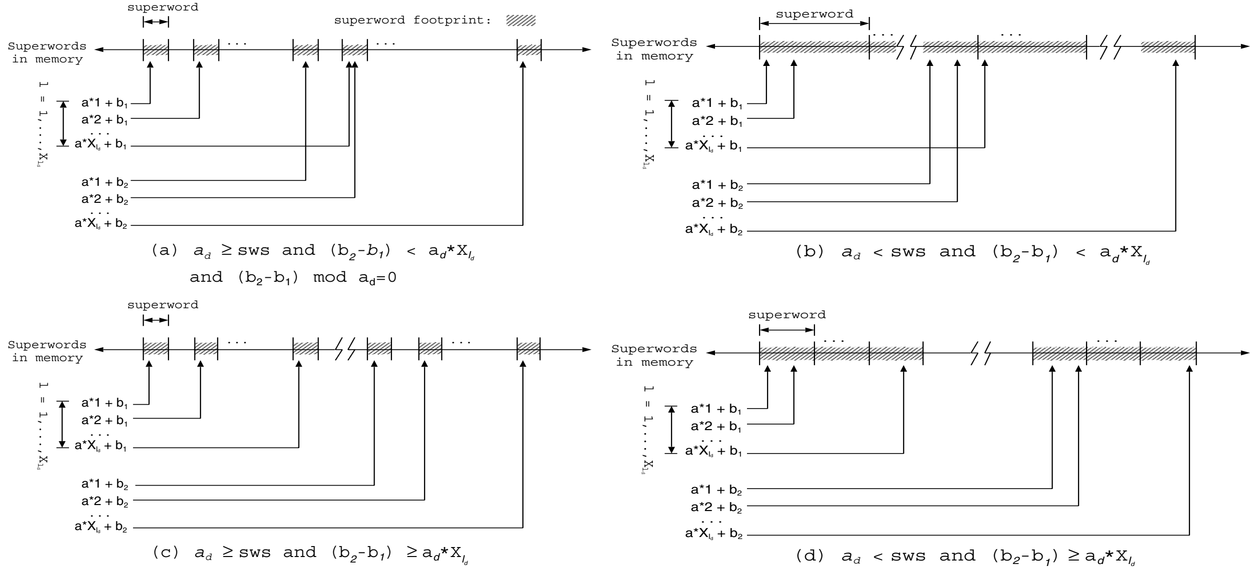


Figure 3. Superword footprint of a group of references.

may overlap in the column dimension if there exist iterations $j_1, j_2, 1 \le j_1, j_2 \le X_j$, such that $j_1 + 2 = j_2$.

The superword footprint of a group V in a set of unrolled loops is computed as follows. For each dimension d, 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 $d < p$, the algorithm partitions references into subsets such that each subset corresponds to a disjoint footprint in dimension d. Then, for each subset, the algorithm recursively computes the footprint in dimension $d+1$, as we now describe.

Dimension d is the lowest dimension ($d = p$). We first compute the group footprint of two array references, and then we extend it for m references. The group footprint of two references $\{v_1, v_2\}$, with lowest dimension subscripts $a_d * l_d + b_1$ and $a_d * l_d + b_2$ such that $b_1 \le b_2$, when loop l_d is unrolled by X_{l_d} is given by Equation (3) in Figure 3.

Equations (3a), (3b) and (3c) correspond to combinations of two basic conditions which determine the superword footprint of a pair of uniformly generated references. The first condition is whether the references have self-spatial reuse within a superword, that is, whether $a_d < sws$. The second is whether the footprints may overlap, which is the case when $(b_2 - b_1) < a_d * X_{l_d}$.

Figure 3 shows four examples of superword footprints corresponding to Equation (3). Figure 3(a) corresponds to Equation (3a), where the footprints may overlap and the group footprint is the union of the two footprints. Each of the individual footprints is a set of X_{l_d} superwords since the references have no spatial reuse. The footprints overlap if $(b_2 - b_1)$ is evenly divided by a_d and there exists an integer value k, $1 \le k \le X_{l_d}$, such that $k = 1 + (b_2 - b_1)/a_d$. This equation precisely computes the overlapped footprint when the two footprints have group temporal reuse. For group spatial reuse, we conservatively approximate the footprint with Equation (3c). In Figure 3(b) the footprints of v_1 and v_2 overlap, and both references have spatial reuse within a superword. The corresponding footprint size is given by Equation (3b).

Figures 3(c) and 3(d) correspond to Equation (3c), where the footprints do not overlap and therefore the group footprint is the sum of the individual footprints. In Figure 3(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 3(d) both v_1 and v_2 have superword spatial reuse.

The number of registers required for reference group $V = \{v_1, v_2, ..., v_m\}$ is computed by extending the equations above to more than two references. Here we describe the most interesting case (corresponding to Equation (3b)), where the footprints overlap and the references have spatial reuse. A subset group $V_i = \{v_{i_{min}}, v_{i_{min+1}}, ..., v_{i_{max}}\}$ is defined by lowest dimension subscripts $a_p * l_p + b_j$,

$i_{min} \le j \le i_{max}$, where the references have been sorted so that $b_{j-1} \le b_j$. V_i has a footprint consisting of contiguous superwords if there is self-spatial reuse ($a_p \le sws$) and possible overlap ($b_j - b_{j-1} \le a_p * X_{l_p}$) for all j such that $i_{min} < j \le i_{max}$. To compute the number of registers required for the entire group, the algorithm partitions V into disjoint subsets V_i as defined above, where $\forall j \quad i_{min} < j \le i_{max}$,

$$\begin{aligned}&(b_j - b_{j-1} \le a_p * X_{l_p}) \wedge \\ &(b_{i_{min}} = b_1 \vee b_{i_{min}} - b_{i_{min}-1} > a_p * X_{l_d}) \wedge \\ &(b_{i_{max}} = b_m \vee b_{i_{max}+1} - b_{i_{max}} > a_p * X_{l_d})\end{aligned} \quad (4)$$

Each subset V_i corresponds to a footprint of contiguous superwords consisting of the union of the individual footprints, with size given by Equation (5).

$$\begin{aligned}F_{l_d}(V_i) &= F_{l_d}(\{v_{i_{min}}, ..., v_{i_{max}}\}) \\ &= \left\lceil \frac{a_p * X_{l_p} + b_{i_{max}} - b_{i_{min}}}{sws} \right\rceil\end{aligned} \quad (5)$$

The total number of superword registers required for the references in V is then the sum of the disjoint footprints of the sets V_i, as in (6).

$$\begin{aligned}F_{l_d}(V) &= \sum_i F_{l_d}(V_i) \\ &= \sum_i \left\lceil \frac{a_p * X_{l_p} + b_{i_{max}} - b_{i_{min}}}{sws} \right\rceil\end{aligned} \quad (6)$$

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

From Section 3.2.1, the footprint of each reference v_i in the unrolled loop body consists of a set of X_{l_d} disjoint footprints, where each of the X_{l_d} footprints starts at superword $(a_d * l_d + b_j) * \prod_{i=d+1}^{p} s_i$, where s_i is the size of dimension i, and $1 \le l_d \le X_{l_d}$.

Therefore the footprints of different references in the group may overlap for some superwords, depending on the values of a_d, b_j and the unroll factor X_{l_d}. The footprints of two uniformly generated references v_1 and v_2 overlap in dimension d if there exists an integer value k such that $1 \le k \le X_{l_d}$ that satisfies Condition 7.

$$a_d * k + b_1 = a_d + b_2. \quad (7)$$

Furthermore, if there exists k satisfying the above condition, the footprints corresponding to the k to X_{l_d} copies of v_1 in the unrolled loop body overlap with those corresponding to the first $X_{l_d} - k + 1$ copies of v_2. The footprint of $\{v_1, v_2\}$ is then given by Equation (8).

$$\begin{aligned}F_{l_d}(v_1, v_2) &= (l_1 - 1) * F_{l_{d+1}}(v_1) \\ &+ (X_{l_d} - l_1 + 1) * F_{l_{d+1}}(v_1, v_2) \\ &+ (l_1 - 1) * F_{l_{d+1}}(v_2)\end{aligned} \quad (8)$$

To compute the size of the entire footprint of V in l_d, our algorithm partitions V into subsets $V_i = \{v_{i_{min}}, ..., v_{i_{max}}\}$ such that, for any j, $i_{min} < j \le i_{max}$, the pair $\{v_{j-1}, v_j\}$ satisfies Condition (4). The footprint of V_i is the union of the overlapped footprints of its reference set and is computed by extending Equation (8) to more than two references.

4 Optimizations for Superword Replacement

After the appropriate unroll factors are determined by the algorithm in the previous section, the unrolled code is then optimized for superword-level parallelism. Not until after SLP are the final code transformations performed to actually exploit reuse in superword registers. In this section, we briefly describe these transformations.

4.1 Replacing Redundant Loads and Stores

Our compiler replaces redundant loads and stores from/to memory with accesses to superword temporaries. Since the code is already unrolled, it is very straightforward to recognize these opportunities. The compiler simply determines 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.

4.2 Packing in Superword Registers

As part of SLP's code generation, whenever data is packed to form superwords, this is done through memory. A data element is loaded into a scalar register from the source location and stored to the destination location. Packing through memory is in some sense motivated by the fact that many multimedia extension architectures do not support register-to-register transfers between scalar and superword register files.

In our system, we have developed an optimization we call *register packing*, shown in Figure 4, to perform this packing in the superword register file. We take advantage of two instructions that are common in multimedia extension architectures, which we call *replicate* and *shift-and-load*. *Replicate* replicates one element of a source register to all elements of a destination register. *Shift-and-load* takes two source registers. The first source register is shifted left by the amount of the third argument and the same amount is taken from the second source register to fill the destination register. Packing these operands in superword registers eliminates numerous scalar loads and stores.

```
w = *((float *)&a + 0);            temp1 = replicate(a, 0);
x = *((float *)&b + 0);            temp2 = replicate(b, 0);
y = *((float *)&c + 0);            temp3 = replicate(c, 0);
z = *((float *)&d + 0);            temp4 = replicate(d, 0);
*((float *)&p + 0) = w;            p = shift_and_load(temp1, temp1, 4);
*((float *)&p + 1) = x;            p = shift_and_load(p, temp2, 4);
*((float *)&p + 2) = y;            p = shift_and_load(p, temp3, 4);
*((float *)&p + 3) = z;            p = shift_and_load(p, temp4, 4);
```

(a) Packing through memory (b) Packing in registers

Figure 4. Register Packing

4.3 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, and illustrated graphically in Figure 5(d). In this example, as before assuming that $sws = 4$, array reference $b[i + j]$ 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 5. As shown in Figure 5(c), only two superwords need to be loaded for the data accessed in the 4 copies of $b[i+j]$ in the loop body, after shifting is applied. Before shifting, $b[i + j]$ had to be loaded from memory (and aligned, for architectures that support only aligned accesses) for each of the four copies of $b[i + j]$ in the loop body.

Detecting the applicability of superword shifting is straightforward, involving checking the dependence distance on the loop for small, constant distances. Code generation is also straightforward, since multimedia extension architectures support efficient shifting and permutation mechanisms for aligning and rearranging data in superwords.

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

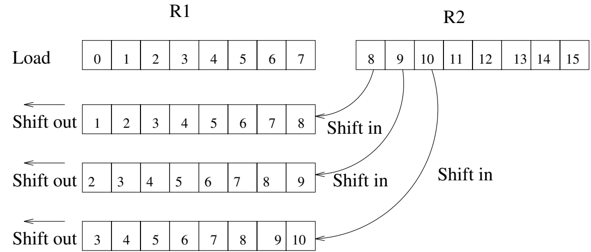
(a) Original loop nest

```
for (i = 0; i < n; i += 4)
  for (j = 0; j < n; j += 4){
    a[i][j:j+3] = b[i+j:i+j+3] * c[j:j+3];
    a[i+1][j:j+3] = b[i+j+1:i+j+4] * c[j:j+3];
    a[i+2][j:j+3] = b[i+j+2:i+j+5] * c[j:j+3];
    a[i+3][j:j+3] = b[i+j+3:i+j+6] * c[j:j+3];
  }
```

(b) After unroll-and-jam and SLP, assuming sws = 4).

```
for (i = 0; i < n; i += 4)
  for (j = 0; j < n; j += 4){
    tmp1[0:3] = b[i+j:i+j+3];
    tmp2[0:3] = b[i+j+4:i+j+7];
    a[i][j:j+3] = tmp1[0:3] * c[j:j+3];
    shift_and_load (tmp1[0:3], tmp2[0:3], 1);
    a[i+1][j:j+3] = tmp1[0:3] * c[j:j+3];
    shift_and_load (tmp1[0:3], tmp2[0:3], 1);
    a[i+2][j:j+3] = tmp1[0:3] * c[j:j+3];
    shift_and_load (tmp1[0:3], tmp2[0:3], 1);
    a[i+3][j:j+3] = tmp1[0:3] * c[j:j+3];
  }
```

(c) After shifting across superword registers.



(d) Graphical depiction of shifting.

Figure 5. Shifting registers for partial reuse.

5 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 [20]. We present a set of results on four multimedia kernels and two scientific applications, derived automatically from our implementation.

5.1 Implementation and Methodology

Figure 6 illustrates the system we have developed for this experiment, which uses the Stanford SUIF compiler as its

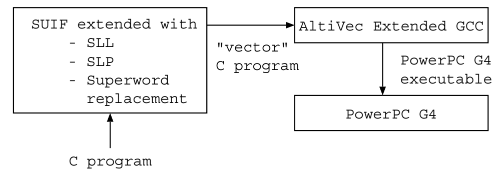
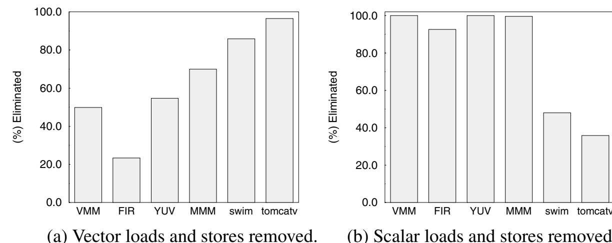


Figure 6. Implementation.

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

Table 2. Benchmark programs.



(a) Vector loads and stores removed. (b) Scalar loads and stores removed.

Figure 7. Reduction in dynamic memory accesses due to superword replacement.

underlying infrastructure [18]. 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[20], and finally, an optimization pass that performs superword replacement as described in Section 4 to steer the compiler to obtain the reuse in superword registers that the SLL algorithm determined was possible.

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.

5.2 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 [20]. As a first step, we verified that we could reproduce their previously reported results. For purposes of comparison, we initially followed the same methodology established in Larsen and Amarasinghe [20]: (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 two sets of results. First, in Figure 7(a), we show the percentage of vector loads and stores eliminated by the full system, as compared with SLP alone. Our approach eliminates over 50% of the vector loads and stores in three of the four kernels, and over 85% in SWIM and TOMCATV. We also eliminate scalar loads and stores using register packing, as described in Section 4. In Figure 7(b), we see that our approach eliminates over 90% of the scalar loads and stores in the four kernels, and over 35% in SWIM and TOMCATV.

Figure 8 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 and Shifting, as described in Section 4. The final bar, entitled Register Packing, shows the additional improvement due to this technique, also described in Section 4.

Overall, we see that in combination, applications achieve speedups between 1.3 and 2.8 over SLP alone, with an average of 2.2X. Consideration of TOMCATV and SWIM shows that both programs have little temporal reuse, al-

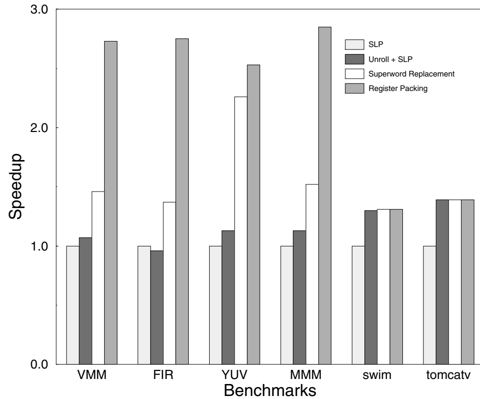


Figure 8. Speedups over SLP alone.

though 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 iteration-space 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 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.

6 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 [13, 6, 31, 32]. 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 [7, 9, 12, 14, 15, 16, 19, 28, 30, 26]. 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 [30, 5, 23]. 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 [4]. 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 [20, 27, 8, 10, 1]. Most, including also commercial compilers [29, 24], are based on vectorization technology [27, 10]. In contrast, Larsen and Amarasinghe devised a superword-level parallelization system for multimedia extensions [20]. 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 [21]. 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.

7 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 2.8X, 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 [27, 8, 10, 1], 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.

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

[2] 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.

[3] 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.

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

[5] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software—Practice and Experience*, 24(1):51–77, 1994.

[6] 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*, pages 252–262, San Jose, California, October 1994.

[7] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *International Conference on Supercomputing*, pages 492–499, 1999.

[8] G. Cheong and M. S. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.

[9] 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.

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

[11] 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*, pages 30.6.1 – 30.6.4, 1992.

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

[13] 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*, pages 328–343, Santa Clara, California, August 1991.

[14] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *TOPLAS*, 17(4):561–575, July 1995.

[15] 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.

[16] 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*, pages 228–239, San Jose, California, October 1998.

[17] 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*, November 1999.

[18] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[19] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimization of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, 1991.

[20] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC Canada, June 2000.

[21] 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.

[22] R. Lee. Subword parallelism with max2. *IEEE Micro*, 16(4):51–59, August 1996.

[23] A. Fernandez M. Jimenez, J.M. Llaberia and E. Morancho. Index set splitting to exploit data locality at the register level. Technical Report UPC-DAC-1996-49, Universitat politecnica de Catalunya, 1996.

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

[25] 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.

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

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

[28] 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, November 1993.

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

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

[31] 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*, pages 30–44, Toronto, June 1991.

[32] Michael J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989.