H-0146 November 17, 2002
Computer Science/Mathematics

IBM Research Report

Compiler Vectorization Techniques for a Disjoint SIMD Architecture

Dorit Naishlos, Marina Biberstein, Ayal Zaks
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Compiler Vectorization Techniques for a Disjoint SIMD Architecture

Dorit Naishlos
IBM Research Lab in Haifa
Haifa University, Israel
dorit@il.ibm.com

Marina Biberstein
IBM Research Lab in Haifa
Haifa University, Israel
biberstein@il.ibm.com

Ayal Zaks
IBM Research Lab in Haifa
Haifa University, Israel
zaks@il.ibm.com

ABSTRACT

This paper presents compiler technology that targets a novel low-power Digital Signal Processor (DSP) architecture. The architecture is characterized by the exploitation of data and instruction level parallelism, and uses a large register file with dynamically composed vectors for data manipulation. We describe how an optimizing compiler can make use of the vector register file with its flexible addressing to efficiently support a range of data access patterns that are present in the digital processing application domain. We describe new challenges presented by this novel DSP architecture, as well as new opportunities for aggressive yet low-overhead optimizations that it introduces. Experiments show that an optimizing compiler can target such an architecture efficiently to achieve performance that is comparable to the optimal hand-generated code for key benchmarks. The resulting compiler technology represents an advance of the state-of-the-art in the area of DSP compilation.

Categories and Subject Descriptors

D.3.4 [**Processors**]: compilers, optimization; C.1.1 [**Single Data Stream Architectures**]: RISC/CISC, VLIW architectures; C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms

Performance, Algorithms

1. INTRODUCTION

The digital processing domain is characterized by very high computing needs coupled with very low power consumption requirements. Typical digital signal processing computations operate on sets of data elements (vectors), that correspond to the digital representation of signals in the analog domain. Such computations often perform the same operation on each element of a vector. Digital Signal Processor (DSP) architectures are tuned to achieve efficient solutions for this application domain.

Contemporary DSPs (such as StarCore SC140[6] and TI C6x[9]) and multi-media extensions to general-purpose processors (such as Intel MMX [20] and Motorola AltiVec [7]) focus on exploiting the natural parallelism present in the applications by including features such as simultaneous execution of multiple instructions (ILP — Instruction Level Parallelism) and simultaneous execution of the same instruction on multiple data elements (SIMD — Single Instruction Multiple Data). In particular, the SIMD features operate on subsets of data elements packed into a "vector" register.

Optimizing compilers use vectorization techniques in order to exploit the SIMD capabilities of the architecture. Such techniques reveal temporal and spatial locality in the scalar source code, often by performing various loop transformations, and then transform groups of scalar "local" instructions into vector instructions. However, the code generated by most DSP compilers is still significantly less efficient than hand-optimized code, even though much effort has been invested in past years to devise advanced compilation techniques for existing DSP architectures. As a result, large portions of applications are being optimized by hand, which is a slow, expensive and error-prone process.

There are various reasons why traditional optimizing compilers cannot reach near-optimal performance for DSP architectures. They mainly stem from having scarce resources in the architecture with tight inter-dependencies. One of the most difficult problems for vectorization is imposed by the memory architecture, which typically provides access to contiguous sequence of memory items. In addition, this sequence is often restricted to start at naturally aligned addresses. The order in which data elements are needed for computation, however, may be neither contiguous nor adequately aligned. The use of traditional SIMD architectures involves additional packing and unpacking complexity, for concatenating data elements into registers and back. This is often done by special instructions that gather, scatter and permute the data elements as needed.

The eLite DSP of IBM [16] was explicitly designed to be compiler-friendly by providing multiple resources with minimum inter-dependencies and irregular constraints, under strict low-power considerations. In this paper we focus on unique architectural features invented in the eLite project to

Submitted to PLDI '03 San Diego, California USA

support efficient vectorization, and show how these features can be utilized by a compiler to provide an effective solution to the problems discussed above.

The architecture of the eLite DSP contains a large vector-element register file, indirectly accessible through vector-pointers which provide efficient solutions to data-reordering and register-renaming problems. This indirectly accessible vector register file supports SIMdD instructions — Single Instructions that operate on Multiple *disjoint* Data. The standard SIMD variation, SIMpD (Single Instructions that operate on Multiple *packed* Data), is also supported in eLite. We show how a range of access patterns can be vectorized with very small constant overheads using SIMdD instructions. The large vector register file together with indirect addressing encourages data reuse, and in particular high-granularity reuse — across loops and loop nests, thus introducing new opportunities for optimizations as well as new challenges that have not been traditionally related to the vectorization domain.

The contributions of this paper are as follows:

- A new compiler optimization for SIMdD architectures that efficiently vectorizes access patterns not necessarily contiguous or properly aligned in memory, with very small constant overheads.
- A new technique for exploiting large vector register files with rotating and indirect addressing, allowing a vectorizing compiler to aggressively exploit temporal and spatial reuse and efficiently hide memory latencies.
- Automatically derived experimental results that compare the performance of compiler-generated code with hand-optimized code. Our results show that the compiler can achieve comparable results to hand-optimized code.

The paper is outlined as follows: Section 2 provides the required background on the target architecture and the compiler. Section 3 shows the unique solution of the eLite DSP to the data reordering problem. Section 4 describes how powerful compiler-controlled caching can be implemented to facilitate reuse and eliminate or hide memory latencies, using innovative vector pointer register. In section 5 we present experimental results that demonstrate the impact of the different optimizations and the competitiveness of the compiled code compared to hand-optimized assembly, and Section 6 concludes.

2. BACKGROUND

The compilation technology described in this paper relies on certain architecture features and is guided by the target domain, i.e., kernels of applications in the areas of line and wireless communications, and voice processing. We provide more details on the computational characteristics and data access patterns of such applications in Section 5. We now give an overview of the SIMdD architecture targeted by our compiler, and the general structure of the compiler with its vectorization component.

2.1 The eLite DSP architecture

The eLite DSP [16] has a multiple-issue statically scheduled architecture, that supports parallelism by executing multiple instructions packed in long-instruction-words (LIW), in conjunction with single instructions operating on different registers at the same time (SIMD). In this paper we focus on the vector units; for more information on the architecture the reader is referred to Moreno et al. [16]. The eLite DSP has four separate functional units that operate on `VL`-element vectors in SIMD fashion (where `VL` denotes the vector-length currently set to 4): Vector Accumulator Unit (VAU), Vector Element Unit (VEU), Vector Pointer Unit (VPU) and Vector Mask Unit (VMU). Each vector unit is associated with its own register file and operations, as follows:

VAU — performs SIMpD operations on (packed) 4-element vector accumulator registers.

VEU — performs SIMdD operations on 4 vector-element registers stored in a large multi-ported register file — the Vector Element Register File (VEF). Every access to the VEF is performed indirectly using Vector Pointer Registers (VPRs). Each Vector Pointer Register contains four elements which serve as indices into the Vector Element Register file, providing access to four arbitrary elements.

VPU — operates on the VPRs. In addition, whenever VPRs are used to access the VEF, they can be incremented implicitly by a predefined amount, optionally wrapping-around to implement circular addressing within a predefined range of the VEF.

VMU — operates on vector mask registers, which can be used analogously to predicate registers in scalar code.

This novel indirect register addressing mechanism enables dynamic composition of vectors, and provides the basis for the development of innovative compiler vectorization techniques.

2.2 The compiler

The eLite DSP compiler has evolved jointly with the architecture, both guided by the performance evaluation of characteristic benchmarks. The eLite DSP compiler is based on Chameleon, the IBM VLIW Research Compiler originally designed for tree-VLIW processors [15]. Chameleon uses an enhanced form of Dependence Flow Graph (DFG) [21] for its internal representation, and also uses static single assignment (SSA, cf. [17]) and reverse-SSA forms extensively. It has a repertoire of standard SSA-based optimizations, and provides a rich infrastructure for compiler development. The dependence-flow graph undergoes a number of transformations, including a novel vectorization phase which is the subject of this paper, followed by register allocation and scheduling, and finally emitting the assembly code.

2.3 The general vectorization scheme

A vectorization phase (called vectorizer) was added to the compiler to make efficient use of the SIMD, and aspecially SIMdD, instructions available on the target architecture.

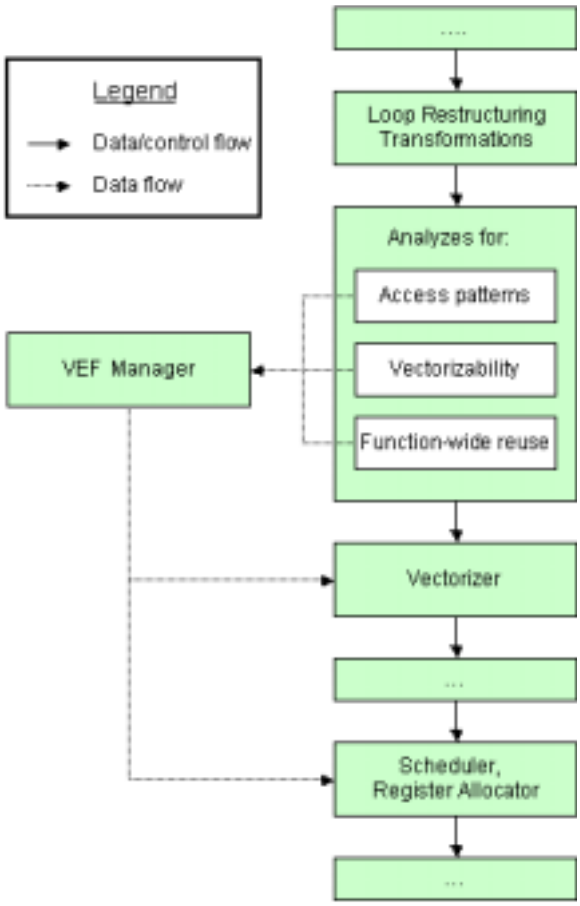


Figure 1: Compilation with SIMDd Vectorization

The main objective of this phase is to extract as much data parallelism as possible and to transform scalar code to SIMD instructions, while enabling subsequent optimization phases to improve the code further. Data and memory dependencies that are present in the generated vectorized code are modelled accordingly and relaxed as much as possible to facilitate following compilation phases — register allocation, scheduling, software pipelining, and other optimizations.

An outline of the vectorization process is given in Figure 1. High-level loop transformations should be applied prior to the vectorizer in order to simplify control structures, expose reuse opportunities, and form vectorizable loops if possible. Such transformations are described in Wolfe [26], and are not in the scope of this paper. The vectorizer contains an analysis stage that identifies vectorizable loops, focusing on inner-most and doubly nested loops. In addition to the standard tests that include memory and data dependence analysis ([26]), and architecture-specific analyses that help guide the vectorization process (e.g., identification of reduction forms), we perform the following analyses:

Data Access Pattern Analysis Identify the data access patterns and memory alignment information. These are needed in order to compute the minimum resources required for vectorizing each loop, and also for setting up vector pointers.

Global Reuse Analysis Identify the set of memory locations accessed in each loop by leaf-to-root propagation through the loop hierarchy. This reveals reuse opportunities both across different iterations of the same loop and across loops.

Guided by the information provided by the analysis phase, the Vector Element File (VEF) Manager selects the schemes for data reorganization (see Section 3) and data reuse (see Section 4), and designates VEF areas to be used by each scheme. The Vectorizer then performs the actual code transformation which includes unroll-and-jam [3]. Finally scalar operations are replaced with vectorized counterparts, and the DFG is updated accordingly. Subsequent scheduling phases that take place after vectorization may continue to interact with the VEF manager in order to improve the schedules by removing VEF anti-dependencies. This depends on VEF availability, and imposes no changes to vector instructions nor unrolling of loops, but only minimal changes to vector pointer setup (see Section 4). The next sections describe the pattern analysis, reorganization schemes, and VEF application for data caching in further detail.

3. DATA REORGANIZATION

Many DSPs and multimedia extensions contain "packed" SIMD (SIMpD) units, where the vector instructions operate on ordered sets of register-packed data for both input and output. Such architectures are vulnerable to data reorganization problems, which occur when the input to one vector instruction is a permutation of another vector instruction's output, or a combination of outputs produced by several vector or scalar instructions. The memory architecture may also cause data reorganization problems, when the order of input (or output) to a vector instruction is not directly supported by memory operations. Situations that require data reorganization are widespread in the context of DSP applications.

In this section we discuss solutions to data reorganization problems in SIMpD and SIMdD architectures. We also present compilation techniques that implement a range of vector reordering patterns at a cost of only a small number of additional SIMdD operations.

3.1 Reorganization in SIMpD Architectures

In SIMpD, the traditional sub-word SIMD architectures, vector instructions operate on vectors that are packed into vector registers. Such architectures provide capabilities to pack and unpack data to and from vector registers. These capabilities are often provided within memory operations [1, 5]. Reordering data in conjunction with memory operations enables packing and unpacking vectors by going through memory. Most architectures also provide vector register copy instructions that permute the data, or instructions that merge two vectors [1, 7, 20, 18]. Such instructions can be used to pack and unpack vectors directly, without going through memory [24].

This approach for reorganizing vector data has a number of drawbacks. If the desired reordering cannot be associated with an appropriate memory operation, there is an overhead of additional vector permute or merge instructions. On the

other hand, if an ordering is associated with a memory operation, it will usually increase its latency, which is relatively long to begin with. Moreover, reordering within memory operations may cause a series of cache misses when addressing several remote locations, and it does not exploit group spatial reuse [26]. If the same data is reused by two vector instructions in different orders, additional permuting instructions are neccesary. Also, if reordering is applied repeatedly (e.g., within a loop), a performance penalty is incurred each time.

3.2 Reorganization in SIMdD and eLite

In contrast to SIMpD architectures, the vectors manipulated by SIMdD instructions are composed "on the fly". In the eLite DSP this is achieved by vector pointers that contain `VL` (Vector Length) independent indices into the vector element file. This architectural feature allows permuting the access without actually moving any data. After setting up a vector pointer according to one vector instruction, subsequent instructions may be able to reuse this pointer at no additional cost by using implicit update capabilities. Thus a one-time penalty for defining the permutation is incurred when setting up the vector pointer, instead of having a penalty at each access.

The compiler realizes this approach to data reorganization in SIMdD using vector pointers, as follows. The techniques apply to a general vector-length; we use `VL`=4 here for simplicity. A vector instruction may have one or more references to vectors, which serve as input or output. For a vector $v = (v_1, v_2, v_3, v_4)$ define the *starting point* $\mathtt{sp}(v)$ to be the VEF index of the first vector element v_1, and define the *vector pattern* $\mathtt{vp}(v)$ to be the 3 offsets (d_1, d_2, d_3) between the consecutive pairs of elements v_i, v_{i+1}. The elements of a vector v with starting point $\mathtt{sp}(v)$ and pattern $\mathtt{vp}(v) = (d_1, d_2, d_3)$ are located in the VEF at positions

$$\{\mathtt{sp}(v), \mathtt{sp}(v) + d_1, \mathtt{sp}(v) + d_1 + d_2, \mathtt{sp}(v) + d_1 + d_2 + d_3\}.$$

If the patterns $\mathtt{vp}(v)$ and $\mathtt{vp}(u)$ of two vectors v and u are identical, the distances between every pair of corresponding elements (v_i and u_i) are the same, and are equal to $\mathtt{sp}(v) - \mathtt{sp}(u)$; in such cases we define this value to be the *distance* $d(v, u)$ between the two vectors.

When setting up a vector pointer, a starting point and vector pattern are specified. In addition, a distance and wrap-around parameters may be given, to be used for implicit updates. For example, setting a vector pointer according to the consecutive pattern $(1, 1, 1)$ with distance 4 and starting position `sp`, abbreviated as $[\mathtt{sp}(1, 1, 1)4]$, provides sequential access to the VEF starting from `sp`.

The same vector pointer can be used by two references, if they refer to vectors with identical patterns. This is because the pointer can be updated (implicitly or explicitly) according to the distance between the two vectors. Furthermore, a series of references to vectors with identical patterns can share the same vector pointer with no additional cost, if all non-zero distances between consecutive pairs are the same. The goal is therefore to partition the set of vector references into separate groups, each assigned a single vector pointer. Similar partitions of scalar accesses to memory have been studied in the past [14, 13]. Figure 2 outlines the algorithm

1. Identify a set of references to vectors in the VEF;
2. Classify the references into ordered groups such that:
   - all references in a group have the same pattern, and
   - the non-zero distances between pairs of consecutive references in a group, according to execution order, are the same.
3. For each group:
   - calculate the vector pointer setup parameters;
   - setup the vector pointer for the first reference;
   - wire the data dependencies among the references in the group.

Figure 2: High-level algorithm for vector pointer setup

that realized this efficient approach to data organization via vector pointers.

Having powerful and efficient data-reorganization facilities for SIMdD instructions eliminates the need for special vector scatter/gather or permute instructions, and also has an important advantage for the memory architecture: it may be sufficient to provide only the most efficient data access patterns to memory. For instance, the memory architecture in the eLite DSP provides access to `VL` consecutive elements in memory, properly aligned on `VL` boundary, with an option to disable storing any subset of elements. In the detailed examples below we show how vector operations on non-contiguous or non-aligned data can still be performed efficiently.

3.3 Reordering via Vector Pointers: example

In DSP applications, the most widespread non-consecutive vector access pattern is probably the constant-stride pattern (d, d, d), in particular for $d = 2$. This pattern is often encountered in DSP computations that deal with complex numbers, where the real and imaginary parts are interleaved in the same input or output array. The complex data needs to be "de-interleaved" in order for SIMD-style computations to be carried out, if different instructions apply to the real and imaginary parts. The results may then need to be "interleaved" back. Constant-stride patterns appear in other contexts as well, such as in decoders and encoders for interleaved codes, or in computations on "very long" data types not supported directly by the architecture.

Consider for example the complex inner-product computation shown in Figure 3. A standard approach to the vectorization of this example uses four SIMD vector multiplications, followed by one vector subtract and three vector add operations, and finally two reduction operations to produce the scalar results (see Figure 4). We assume for clarity that `len` is known to be divisible by `VL`. Otherwise, standard unrolling and memory misalignment techniques (described below) are used. We next examine how the data is loaded into the VEF and how it is accessed by the multiplications.

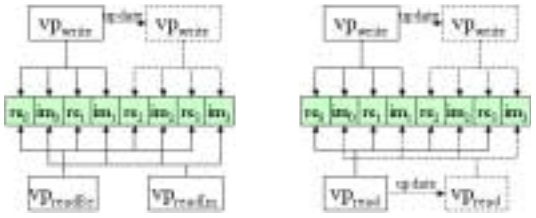
Assume at first that the input arrays are properly aligned on `VL` boundaries; this assumption is later relaxed. Two loads are needed in order to bring 4 elements of `re1` needed for

```
void inner_product(int len, short *a1, short *a2,
                   short *re_result, short *im_result){
    short re1, im1, re2, im2;
    short re1re2, re1im2; im1re2; im1im2
    short re, im;
    short accre = 0;
    short accim = 0;
    for(int i = 0; i < len; i++){
        re1 = a1[2*i];
        im1 = a1[2*i+1];
        re2 = a2[2*i];
        im2 = a2[2*i+1];
        re1re2 = re1*re2;
        re1im2 = re1*im2;
        im1im2 = im1*im2;
        im1re2 = im1*re2;
        re = re1re2 - im1im2;
        im = re1im2 + im1re2;
        accre = accre + re;
        accim = accim + im;
    }
    *re_result = accre;
    *im_result = accim;
}
```

Figure 3: Inner product of complex vectors

```
void v_inner_product(int len, short *a1, short *a2,
                     short *re_result, short *im_result){
    short re1[1:4], im1[1:4], re2[1:4], im2[1:4];
    short re1re2[1:4], re1im2[1:4]; im1re2[1;v]; im1im2[1:4]
    short re[1:4], im[1:4];
    short accre[1:4] = 0;
    short accim[1:4] = 0;
    for(int i = 0; i < len; i+=4){
        re1[1:4] = a1[2*i,2*i+2,2*i+4,2*i+6];
        im1[1:4] = a1[2*i+1,2*i+3,2*i+5,2*i+7];
        re2[1:4] = a2[2*i,2*i+2,2*i+4,2*i+6];
        im2[1:4] = a2[2*i+1,2*i+3,2*i+5,2*i+7];
        re1re2[1:4] = re1[1:4]*re2[1:4];
        re1im2[1:4] = re1[1:4]*im2[1:4];
        im1im2[1:4] = im1[1:4]*im2[1:4];
        im1re2[1:4] = im1[1:4]*re2[1:4];
        re[1:4] = re1re2[1:4] - im1im2[1:4];
        im[1:4] = re1im2[1:4] + im1re2[1:4];
        acc_re[1:4] = acc_re[1:4] + re[1:4];
        acc_im[1:4] = acc_im[1:4] + im[1:4];
    }
    *re_result = reduce2sum(acc_re[1:4]);
    *im_result = reduce2sum(acc_im[1:4]);
}
```

Figure 4: vectorized inner prod. of complex vectors



(a) Standard **(b) VP-economic**

Figure 5: Reordering at VEF read

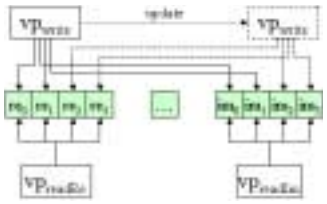


Figure 6: Reordering at VEF write

each iteration, such as: `a1[0]`,`a1[2]`,`a1[4]`,`a1[6]`, because one load can only access 4 elements that are aligned and consecutive in memory. However, these same two loads also bring 4 elements of `im1`. One possibility for vector pointer setup is depicted on Figure 5a. Eight consecutive elements of each array (half of which are real and half imaginary) are loaded into consecutive locations in the VEF. For example, elements `a1[0]`,`a1[1]`,...,`a1[7]` are loaded into vector-element registers $0, \ldots, 7$, respectively. This provides all the data needed for the four multiplications (in a single iteration), and can be done by two loads sharing the same vector-pointer ($\mathrm{VP}_{\mathrm{write}}$) setup to $[0(1,1,1)4]$ — a consecutive pattern starting at element 0.

Each vector multiplication instruction has references to two vectors, one from each input array. Assuming the data was loaded into the VEF as described above, the vector patterns of these references are $(2,2,2)$. There are several alternatives for assigning vector pointers to these references. One possibility is to assign four pointers, one for each part (real/imaginary) of each input array, using the following setups: $[a1[0](2,2,2)8]$, $[a1[1](2,2,2)8]$, $[a2[0](2,2,2)8]$ and $[a2[1](2,2,2)8]$. Referring to Figure 5a, $[a1[0](2,2,2)8]$ corresponds to $\mathrm{VP}_{\mathrm{readRe}}$, and $[a1[1](2,2,2)8]$ corresponds to $\mathrm{VP}_{\mathrm{readIm}}$. In each iteration, each of these pointers will be used once without update, and then once with implicit update of distance 8.

More vector pointers can be used (if available) to eliminate dependencies and produce more regular code: each pointer can be split into two pointers that are updated after every use. On the other hand, the number of vector pointers can also be reduced if desired: two pointers to real and imaginary parts of the same input array can be merged into one pointer, because they have identical patterns (see Figure 5b). By merging two pairs of pointers, a minimum of two vector pointers suffices. However, the merged pointers will need to be updated by at-least two different distances. This demonstrates the various alternatives to consider when assigning vector pointers, and the associated tradeoffs.

The above vector pointer setups used a consecutive pattern for loads and reordered patterns for accessing the VEF. This way, elements appear in the VEF in the same order as they appear in memory, which may be useful for debugging and other purposes. It is possible to reorder the data as it is loaded into the VEF, so that it can then be accessed in a consecutive pattern. Referring to Figure 6, the pattern of the vector pointer $\mathrm{VP}_{\mathrm{write}}$ used by the load is $[a1[0](x, 1-x, x)2]$, where $x = a1[1] - a1[0]$. This is possible in general for constant-stride patterns $(d, d, \ldots, d)$ where d divides `VL` , and corresponds to a VEF allocation containing d areas of size `VL` each rather than a single area of size d*`VL` that is required by the original approach. Such a disjoint allocation might be preferred, depending on VEF availability.

Reordering data to be stored from the VEF into memory is done along the same lines. In general, the input for SIMdD instructions in eLite can originate from memory (as in the above example), from the output of SIMpD instructions, or already reside in the VEF. In the latter case, the vector patterns are determined directly by the SIMdD instructions according to the location of the data in the VEF. In the first two cases however, there is freedom in choosing a VEF allocation scheme and the corresponding vector patterns for transferring the data into the allocated VEF indices and out to the SIMdD instructions, in a way that addresses performance requirements and resource limitations.

The complex inner-product example contains temporal and spatial reuse, where all the vector patterns are identical. There are cases where several distinct vector patterns are used, all referring to the same data. One such example involves squaring a matrix, which requires accessing both its rows and its columns. Vector pointers can be used to implement such multiple accesses efficiently, again without reordering the data itself.

3.4 Handling Alignment Constraints

Memory alignment constraints raise problems which are related to data reordering. Accessing a block of memory from a location which is not aligned on a certain boundary is often prohibited or bears a heavy performance penalty. Techniques used to avoid these penalties such as loop-peeling [2] or dynamic alignment detection [11] do not always work and increase code size. Techniques that try to confront this problem usually incur a penalty that grows linearly with the data set size [12, 5].

Our compiler handles data alignment as a special case of data reordering, where the access to a contiguous data set is slightly shifted in order to comply with the memory alignment constraints. For example, if the input arrays in Figure 3 are not aligned (or not known to be aligned), we proceed as follows. Vector loads are generated as if the arrays start at the nearest preceding aligned address and end at the nearest succeeding aligned address. This is accomplished by having one extra load in the loop prologue, and requires a few extra spaces in the VEF, but the vector pointer pattern remains the same. The vector pointers used to access the VEF also retain the same pattern, but skip over the first few loaded elements.

Stores to unaligned data are handled similar to loads, except that the first and last stores are masked appropriately so as not to write past the bounds of the target array. Thus, we are able to handle loads and stores into unaligned locations using accesses to aligned memory only, paying only a small constant performance penalty.

4. DATA MANAGEMENT

The previous section showed how data reorganization problems can be solved efficiently with the use of vector pointers. This section shows how vector pointers can also help improve performance through the exploitation of spatial and temporal reuse, if vector data is managed appropriately. Indeed, exploiting reuse opportunities aggressively is known to have dramatic effects on application performance, particularly because it can eliminate or reduce the overhead associated with long memory access latencies. This is especially true for DSP and multimedia kernels, characterized by having tight loops that perform well-structured computations on single- or multi-dimensional arrays.

4.1 Data Management: Existing Solutions

The classic solution to the problems of long memory latencies is aggressive usage of large multi-levelled cache hierarchies. Caches are very effective for most-frequently used data, but scientific computations which often lack temporal locality of accesses may still spend more than half of their execution time stalled on memory requests [25]. Much work has been done in the past on detecting temporal and spatial locality within loops, and exploiting them using various loop transformations to improve hardware cache performance [26]. Recent work also focused on inter-nest locality [10].

A complementary approach uses special *prefetching* instructions, generated either by a compiler or by the hardware at runtime, that attempt to anticipate future accesses [8, 25]. Such techniques address the latency hiding problem, rather than data reuse issues.

Various hardware mechanisms were proposed to augment the cache performance, especially for streamed applications [19, 4, 22]. Most of these mechanisms are not controlled by software and cannot benefit from the data flow information available to the compiler. Such information can be utilized by compiler-controlled caching, which requires loop unrolling [24] or other solutions to register naming problems [8]. In contrast, the SIMdD and VEF available in the eLite DSP can support aggressive compiler-controlled caching, without requiring loop unrolling, as discussed next.

4.2 Cache Management in the VEF

The large number of registers available in eLite's vector element file, together with the indirect access using vector pointers, are ideal for implementing a software-managed vector-data cache. However, the latency of accessing the VEF is greater than that of ordinary (scalar or packed vector) registers. As a consequence, very simple benchmarks such as vector addition are vectorized more efficiently using traditional SIMpD instructions and packed vectors, rather than utilizing the VEF. In general, vectorizing a computation using SIMdD instructions that access the VEF is preferred to using SIMpD instructions if any of the following holds: (1) data reorganization is needed, as described in Section 3; (2) unit assignment considerations — these are not in the scope of this paper; (3) data reuse can be exploited.

Spatial and temporal reuse can be exploited to hide or reduce the overhead of loading data into the VEF, making the use of SIMdD more attractive. Such reuse occurs, for example, when (all or part of) the data required by a computation already resides in the VEF, because it was needed or put there by a previous computation. In other cases only a few loads that are scheduled early may suffice to provide all the data required by a computation. In all such cases, the key issue is to be able to move loads early, possibly even to loop prologues, and then merge overlapping loads. In order to load ahead of time ("pre-load") one needs a large storage. However, in order to use preloaded data efficiently, one also

```
    cnt0 := 4
loop:
    i0 := load(i5++)
    i10 := i10 + i0
    branch loop if (cnt0-- != 0)
```

Figure 7: A simple loop example

needs rotated addressing, as explained below. Interestingly, the VEF with its vector pointers supplies these needs in a new and powerful way.

4.3 Rotating Vector Registers

The overhead of load operations can often be eliminated by passing values in registers instead of through memory, or hidden by scheduling loads early. A major difficulty with this approach is register renaming (cf. [17]). Consider, for example, a simple loop for summing the elements of a vector, depicted in Figure 7. Suppose that in this example, the data that resides in addresses `i5`. . . `i5+3` can be passed in registers `i0, i1, i2, i3`. In order to use these registers, the compiler would typically need to unroll the loop, which might seriously affect code size. Another solution is to perform register rotation, either by software [3] or by special hardware [8]. If loads cannot be eliminated, their latencies may be hidden by scheduling them early, in a software-pipelined fashion. Here again, several values are kept "alive" at the same time, requiring more than one physical register and hence duplication of code or rotation of registers.

The indirect register addressing of eLite's vector pointers solves this naming problem by providing rotating vector-register addressing, that is much more powerful than existing rotating (scalar-)register mechanisms. Each vector-pointer defines the set of vector-elements over which it rotates, independent of other vector-pointers. The elements over which a vector pointer rotates need not be consecutive; indeed, two rotating vector pointers may have some elements in common. The rotation itself is activated for each vector pointer independently, and is not associated with any global instruction. Furthermore, changing the number of registers over which to rotate is accomplished by simply changing a parameter upon vector pointer setup, rather than reassigning registers to all relevant instructions. This is an important advantage, because such changes may be applied late or even after the vectorization phase takes place.

Consider for example a vector pointer that rotates over the same `VL` elements. Such a vector pointer creates anti dependencies which might severely constrain instruction scheduling and software pipelining. Increasing the number of elements over which the vector pointer rotates to, say, 2*`VL` , will increase the distance of these dependencies and enable aggressive code motion. Such decisions may be taken during scheduling (i.e., after vectorization) according to the VEF availability, or during vectorization in anticipation of future scheduling needs.

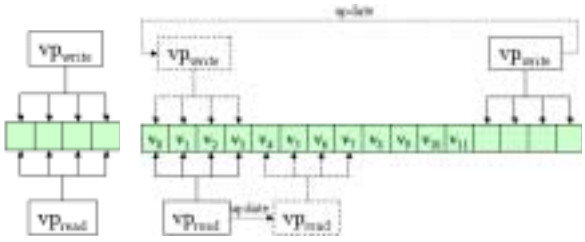
Another very important advantage of using vector pointers for rotating addressing, is that the operations writing to rotated registers are "disengaged" from the operations

```
int accumulate(int len, short *a){
    short tmp;
    int result = 0;
    for(int i = 0; i < len; i++){
        tmp = a[i];
        result += tmp;
    }
    return result;
}
```

Figure 8: Vector accumulation example

```
int v_accumulate(int len, short *a){
    short v_tmp[1:4];
    int v_result[1:4] = 0;
    int result;
    for(int i = 0; i < len; i+=4){
        v_tmp[1:4] = a[i,i+1,i+2,i+3];
        v_result[1:4] = v_result[1:4] + v_tmp[1:4];
    }
    result = reduce2sum(v_result[1:4]);
    return result;
}
```

Figure 9: Vector accumulation: naive vectorization



(a) Naive (b) With pre-loading

Figure 10: Vector pointer setup for v_accumulate

that read from them. This is true for indirect addressing in general. Using vector pointers, however, provides rotating addressing in conjunction with data reorganization, as described in the previous section.

4.4 Pre-loading into the VEF: an example

We illustrate the potential of using vector pointers for rotating addressing by the following example. Consider a function computing the sum of all elements in a vector (Figure 8). The "naive", lazy-loading vectorization of this function is shown on Figure 9, with corresponding vector pointer setup shown on Figure 10a.

For architectures with high load latencies, such an implementation would spend most of its time in the loop waiting for the load to complete. In order to achieve optimal performance, the loads (or the entire loops) should be software pipelined (cf. [23]): several loads should be executed before the loop, each iteration should work on data that was loaded in some previous iteration and fetch data for one of the following iterations. In the eLite DSP this optimization can be performed by simply (1) placing loads in the loop prolog, and (2) extending the VEF area allocated for the array, so that it will be large enough to hold all the

loaded elements until they are last used. The add instruction remains intact; only the setup of the pointer it uses should be modified according to the allocated VEF space. Figure 10b shows the state of the vector pointers at the loop entry, assuming that the number of loads hoisted to the loop prolog is three. After executing these three loads, the vector pointer used by the loads points to the fourth 4-element chunk of the allocated area (assuming again that `VL` =4 for simplicity). In the course of the first iteration, this chunk will be filled with array elements `a[12]`–`a[15]`, while the addition instruction will read elements `a[0]`–`a[3]` provided by the first load. An implicit update with wrap-around set at distance 16 will then rewind the load pointer VP_{write} back to the beginning, so that the following (fourth) load will overwrite the first load with array elements `a[16]`–`a[19]`. Meanwhile the pointer used by the add VP_{read} will continue to the second vector (holding array elements `a[4]`–`a[7]`, and will loop back to the first vector only at the end of the fourth iteration.

The compiler detects such reuse opportunities, and allocates VEF entries to support pre-loading across loop iterations and between loops according to available resources, in order to hide load latencies and improve performance.

5. EXPERIMENTAL RESULTS

This section evaluates the different optimizations discussed in previous sections, and demonstrates the overall competitiveness of the compiler with hand-optimized codes. Being an example of an SIMdD architecture, the eLite DSP provides an excellent test-bed for evaluating the compilation techniques described above.

5.1 Experimental methodology

The experimental results we present in this section were generated automatically using a cycle-accurate simulator and profiler. Code generated by the eLite compiler, incorporating the technology described in this paper, is compared to code optimized for the eLite architecture independently by expert assembly programmers.

Table 1 provides a brief description of the benchmarks used in our experiments. These benchmarks are representative of the main computation kernels in our target application domain. The benchmarks cover a range of access patterns including consecutive (eudist), reverse(rfir-1, rfir-b, xfir-b), unaligned (u-add), strided (xfir-b, dec, inter), and column-wise (v-sad, idct) patterns, both when writing to and when reading from the VEF. Typically a single benchmark features patterns of more than one type. In addition, the benchmarks consist of different loop hierarchies, including single-nest loops (benchmarks numbered 8–12 in the table), doubly-nested loops (benchmarks 1–7) as well as more involved Control Flow Graphs (benchmark 13). As such, these benchmarks reflect different data orderings and reuse opportunities.

5.2 Comparison with hand written code

The high performance demands in the digital processing domain have traditionally required programming DSP's in assembly language. The eLite DSP compiler, using innovative SIMdD compilation capabilities, is attempting to close the gap between compiler generated and hand written codes, as we demonstrate here. Figure 11 displays the relative execution time of compiled codes, normalized to hand-written execution time. In both cases the codes are vectorized using the VEF and the techniques described in this paper.

Name	Description
1 rfir-b	real FIR filter for a block of outputs
2 xfir-b	complex FIR filter for a block of outputs
3 mcc	maximum cross correlation
4 mat	matrix multiply by vector
5 inter	interpolation with up-sampling rate 1:2
6 dec	decimation with down-sampling rate 2:1
7 v-sad	sum of absolute differences for video applications
8 rfir-1	real FIR filter for a single output
9 gather	gather dispersed bits into a vector
10 prod	inner product of two vectors
11 eudist	euclidian distance of two vectors
12 u-add	summation of two unaligned vectors into a third
13 idct	2-D inverse Discrete Cosine Transform

Table 1: Benchmark Description

In most cases the hand-written codes do not make any assumption regarding the problem size except that it is divisible by some amount (usually this amount is `VL`=4). The compiler is given a similar assumption in these cases. The relative results for these cases are denoted as "general data size". In addition, we present the relative performance that is achieved under the assumption that the data set fits into the VEF, an assumption reasonable in the respective application domain. These are denoted as "data fits in VEF". Some benchmarks were originally tailored to a fixed data size; for those we do not present results under the "general data size" category.

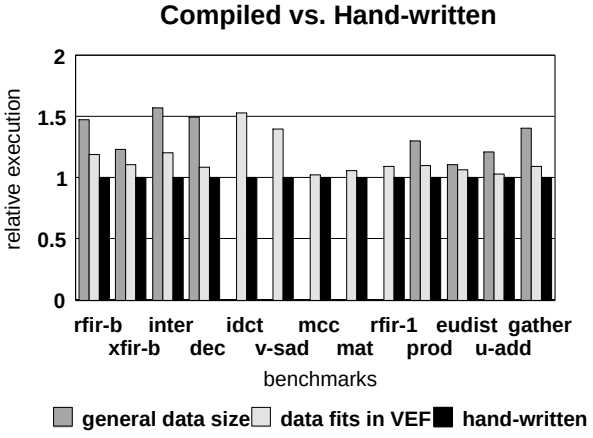


Figure 11: Execution time of compiler generated codes relative to hand written codes

For the general data size case, the average execution time degradation of compiled codes compared to hand-written codes is 35%, with an average of 25% difference for single-nested loop kernels and an average of 44% for multi-nested loop kernels. The main reason for this difference lies in the compiler scheduling scheme which is currently less efficient in exploiting ILP at higher levels of loop hierarchies.

For the case where the data fits into the VEF, the average

performance difference is 15%, with an average of 7.5% for single-nested and 19% for multi-nested loop kernels. The main reason for the improved relative performance for this set of benchmarks is the more efficient scheduling. The worst relative performance for an inner-loop kernel (40%) is obtained by the kernel "gather", where the data size is not assumed to be divisible by 4; at present the epilogue generated by the compiler for handling such cases is less efficient than hand-written code.

5.3 Impact of VEF space usage

As described in Section 4, the rotating registers technique facilitates a pre-loading optimization which utilizes additional VEF area and facilitates more aggressive scheduling. Figure 12 demonstrates the impact of VEF size on performance, by comparing the two extremes: minimal VEF size and maximal VEF size. Minimal VEF size implies that exactly `VL` elements (4 in our case) are reserved for each array, just enough to enable SIMdD vectorization. Maximal VEF size implies that the VEF space reserved for an array is extended as much as possible; if the data size is known at compile time, space for the entire data will be allocated in the VEF (if possible; for the benchmarks at hand, this was the case). If the data size is not known at compile time, the reserved VEF space will be extended by a default amount.

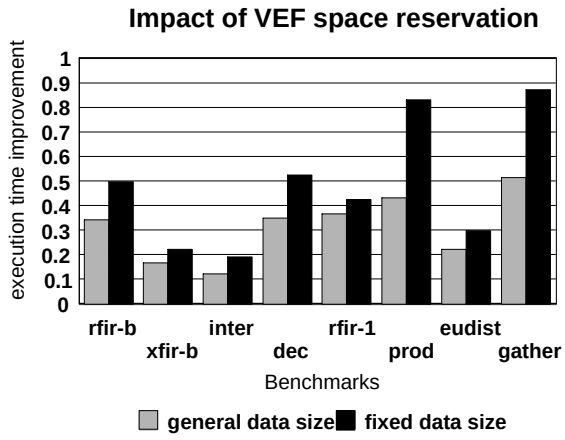


Figure 12: Impact of amount of VEF space reservation on performance

The impact of the pre-loading optimization, as shown in Figure 12, reflects the amount of computation being performed on each element; if a large amount of computation takes place, part of the computation will hide the latency of loading other elements, leaving a pre-loading optimization only little room for further improvement. This is the case in "eudist" for example. However, if very little computation takes place, then most of the code will be stalled waiting for loads to complete, and the impact of pre-loading is therefore increased. This is the case in "prod" for example.

5.4 Impact of data reuse

Computations containing nested loops often introduce the opportunity to exploit data reuse across iterations in an outer nesting level, using the techniques described in Section 4. We have applied these inter-nest techniques to the multi-nested loop kernels, under the assumption that the data set fits into the VEF. In cases where the ranges accessed in different outer-loop iterations overlap, the improvement factors achieved were 27%, 29%, 20%, and 10% in dec, inter, rfir-b and xfir-b respectively. The variation between the different benchmarks is due to the amount of overlap and the relative percentage of load operations in the computations.

The kernels that have column-wise access patterns (idct, v-sad) have no overlap between different iterations, and are therefore not improved by this inter-nest optimization. However, by applying a more aggressive analysis that searches for reuse even higher in the loop-hierarchy tree, additional improvement can be achieved. The column-wise access pattern in idct is encapsulated within another loop, and followed by a similar triply-nested loop; exploiting the data reuse available across these loops and loop nests provides a 42% improvement in execution time.

6. CONSLUSION

This paper presents a set of compilation techniques for an SIMdD architecture. The novel capabilities of the architecture can provide low-overhead and efficient solutions to the traditionally difficult problems of data reordering, data misalignment, and register renaming. We especially focused on applying these techniques to compiler vectorization and data reuse optimizations. Using these techniques, we demonstrated that the performance of the code generated by our compiler is well within a factor of 2 compared to hand-optimized code for a set of benchmarks representative of the DSP domain. For most benchmarks, the degradation did not even exceed 20%.

Work on the vectorizing compiler for the eLite DSP is still in progress. In particular, improved techniques for efficient vector element file (VEF) allocation including prioritization are investigated, to allow the compiler to take full advantage of the VEF resources available when larger applications are considered. The interaction between the VEF Manager and instruction scheduling (including software pipelining) can also provide extra optimization opportunities. In addition, we are exploring the potential for applying the VEF and vector pointer capabilities for additional optimizations, such as vectorization of induction variables and caching of scalar values.

Combining novel architecture capabilities with innovative compiler techniques allows our compiler to seamlessly and and efficiently solve data reorganization problems, and then focus on critical issues of data reuse and efficient scheduling, in the context of SIMD vectorization. This is another major step on the way to a competitive DSP programmable in a high-level language.

7. REFERENCES

[1] K. Asanovic and D. Johnson. Torrent architecture manual. Technical report, ICSI, 1996.

[2] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems. *Intel Technology J.*, February 2001.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *PLDI*, pages 53–65, June 1990.

[4] W. Y. Chen, R. Bringmann, S. A. Mahlke, R. E. Hank, and J. E. Sicolo. An efficient architecture for loop based data preloading. In *Micro*, 1992.

[5] J. Corbal, R. Espasa, and M. Valero. Exploiting a new level of DLP in multimedia applications. In *Intl. Symposium on Microarchitecture*, pages 72–, 1999.

[6] P. D'Arcy and S. Beach. Starcore sc140: A new dsp architecture for portable devices. In *Wireless Symposium*. Motorola, September 1999.

[7] K. Diefendorff and P. K. D. et al. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, March-April 2000.

[8] G. Dohsi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data prefetches with rotating registers. In *PACT*, pages 257–267, 2001.

[9] T. Instruments. www.ti.com/sc/c6x, 2000.

[10] M. Kandemir, I. Kadayif, A. Choudhary, and J. A. Zambreno. Optimizing inter-nest data locality. In *PACT*, pages 127–135, 2002.

[11] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *Intl. J. of Parallel Programming*, 28(4):347–361, 2000.

[12] S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Memo 621, MIT LCS, November 2001.

[13] R. Leupers and F. David. A uniform optimization technique for offset assignment problems. In *11th Int. Symp. on System Synthesis (ISSS)*, 1998.

[14] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Storage assignment to decrease code size. In *PLDI*. ACM SIGPLAN, 1995.

[15] J. H. Moreno, M. Moudgill, K. Ebcioglu, E. Altman, B. Hall, R. Miranda, S. K. Chen, and A. Polyak. Simulation/evaluation environment for a vliw processor architecture. *IBM Journal of Research and Development*, 41(3):287–302, May 1997.

[16] J. H. Moreno, V. Zyuban, U. Shvadron, F. Neeser, J. Derby, M. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. Asaad, T. Fox, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. Reseach Report RC22568, IBM, Sept 2002.

[17] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] H. Nguyen and L. K. John. Exploiting simd parallelism in dsp and multimedia algorithms using the altivec technology. In *Intl. Conf. on Supercomputing*, pages 11–20, 1999.

[19] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design and Test Conf.*, March 1997.

[20] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *IEEE Micro*, pages 43–45, August 1996.

[21] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: an algebraic approach to program dependencies. In *POPL*, pages 67–78, 1991.

[22] M. Postiff. *Compiler and Microarchitecture Mechanisms for Exploiting Registers to Improve Memory Performance*. PhD thesis, U. of Michigan, 2001.

[23] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *PLDI*, pages 1–11, 1996.

[24] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *PACT*, 2002.

[25] S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[26] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.