Vectorizing for a SIMdD DSP Architecture

Dorit Naishlos Marina Biberstein Shay Ben-David Ayal Zaks

IBM Haifa Labs
Haifa University Campus
Haifa 31905, Israel
{dorit, biberstein, bendavid, zaks}@il.ibm.com

Abstract. The Single Instruction Multiple Data (SIMD) model for fine-grained parallelism was recently extended to support SIMD operations on disjoint vector elements. In this paper we demonstrate how SIMdD (SIMD on disjoint data) supports effective vectorization of digital signal processing (DSP) benchmarks, by facilitating data reorganization and reuse. In particular we show that this model can be adopted by a compiler to achieve near-optimal performance for important classes of kernels.

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

Keywords

SIMD, vectorization, subword parallelism, rotating register file, compiler controlled cache, data reuse, viterbi

1. INTRODUCTION

Architectures of modern Digital Signal Processors (DSP) [7, 10] and multi-media extensions [19, 8] aim to combine high performance execution with low power consumption. In particular, they exploit the natural parallelism present in signal processing applications by simultaneously executing the same instruction on multiple data elements. This Single Instruction Multiple Data (SIMD) model usually requires that operands be packed in advance in "vector" registers.

Programmers and optimizing compilers use vectorization techniques [23] to exploit the SIMD capabilities of the architecture. Such techniques reveal temporal and spatial locality in the scalar source code and transform groups of scalar instructions into vector ones. It is often very complicated to apply vectorization techniques to a DSP architecture, because the latter usually has scarce resources with tight inter-dependencies between them. Vectorization is often further impeded by the memory architecture, which typically provides access to contiguous memory items only with additional alignment restrictions. Computations, on the other hand, may access data elements in an order which is neither contiguous nor adequately aligned (e.g. a Complex Filter). Packing data elements into and out of vector registers is usually done with special gather, scatter or permute instructions, which incur additional performance penalties and complexity.

The eLite DSP of IBM [15] was explicitly designed to support efficient vectorization techniques by providing multiple resources with minimum inter-dependencies and irregular constraints, yet under strict low-power considerations. It features a large vector-element register file accessible indirectly through vector pointers, supporting Single Instruction Multiple *disjoint* Data (SIMdD) instructions. Traditional Single Instruction Multiple *packed* Data (SIMpD) instructions that operate on vector registers are supported as well.

In this paper we focus on these unique architectural features and describe how they can support vectorization techniques applied both manually and automatically using an optimizing compiler. We show how SIMdD helps solve data-reordering and register-renaming problems that occur when large amounts of vector data need to be reordered or re-accessed, thereby enabling the vectorization of a wide range of patterns with only a small constant overhead. In particular, we show how a large vector-element file can support data reuse across loops and loop nests, thus introducing new optimization opportunities as well as new challenges that have not been traditionally related to the vectorization domain.

The contributions of this paper are as follows:

- New vectorization techniques for SIMdD architectures that efficiently vectorize non contiguous and/or misaligned access patterns, with very small constant overheads.
- New techniques 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 or eliminate memory latencies.
- SIMD acceleration of Viterbi decoder using a random, indirectly accessible vector register file.

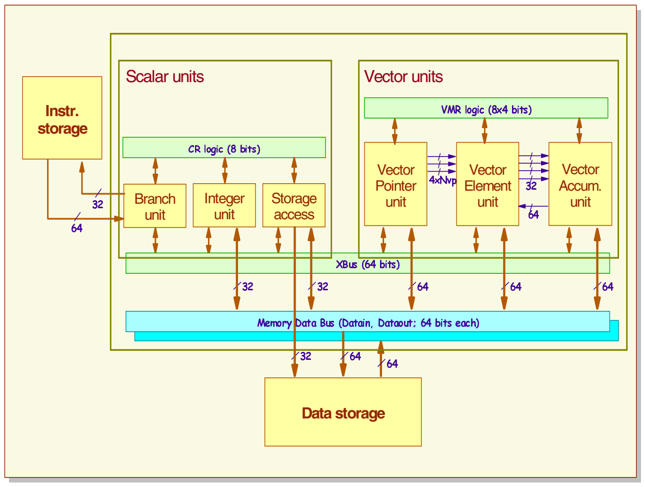


Figure 1: Block Diagram of the eLite Architecture

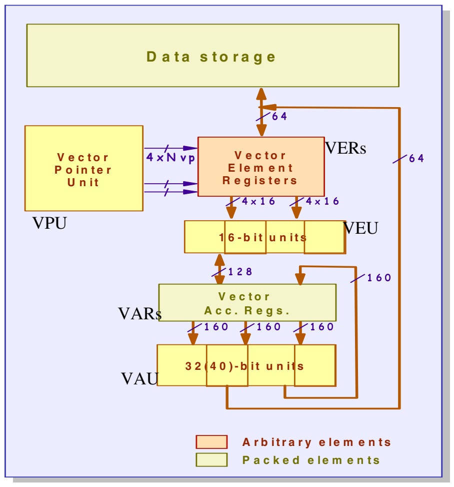


Figure 2: eLite Vector Units and Vector Programming Model

- 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 and Section 3 provide a brief background on the target architecture and the compiler. Section 4 presents two key kernels that will be used throughout the rest of the paper to demonstrate the different programming and compilation techniques. Section 5 describes how compiler-controlled caching can be implemented to facilitate reuse and eliminate or hide memory latencies, using innovative vector pointer registers. Section 6 shows the unique solution of the eLite DSP to the data reordering problem, followed by a demonstration of how the random access pattern is vectorized in Section 7. In section 8 we present experimental results that demonstrate the competitiveness of compiler-optimized code compared to hand-programmed assembly, and Section 9 concludes.

2. AN OVERVIEW OF ELITE

The eLite DSP is a load/store RISC-like architecture. In addition to SIMD parallelism, Instruction-level parallelism is realized in eLite through the packing of multiple instructions into long-instruction words (LIWs). Figure 1 presents a high-level view of the architecture. The units which are most relevant for this paper are:

Vector Element unit: performs SIMdD operations on the data stored in Vector Element Registers. The number of these registers is between 64 and 4096 and is implementation-dependent.

Vector Pointer unit: performs SIMpD operations on 16 Vector Pointer Registers, which are used to access the Vector Element Registers.

Vector Accumulator unit: performs SIMpD operations on data in 16 Vector Accumulator Registers.

In addition, Vector Mask Registers support conditional execution of individual operations within a vector instruction, using dynamically evaluated masks.

The SIMD units are shown in more detail in Figure 2. The Vector Element Unit (VEU) operates on vectors of 4-elements, each element being a 16-bit Vector Element Register (VER). The elements of each vector are selected from the Vector Element File (VEF) — a large, multiport, scalar register file containing $2^{N_{VP}}$ independently addressable elements. N_{VP} is a parameter associated with implementations of the architecture, with an architectural limit value of 12 (4096-element VEF) and typical values for initial implementations in the range of 6 to 9 (64-element to 512-element VEF).

The selection of elements from the Vector Element File (VEF) is indirect, via indices specified by Vector Pointer Registers (VPRs; these are omitted from Figure 2 for simplicity). The vectors operated on by the VEU are thus dynamically composed and can be extensively reused by setting the VPRs appropriately. The VAU on the other hand, operates on "packed" Vector Accumulator Registers (VARs), each holding 4-elements of 40-bit each.

We now describe how Vector Pointers index elements in the VEF. A VPR contains four indices, each in the range of 0 to |VEF|-1. In addition, associated with each VPR is a mechanism for post-incrementing the indices with modulo wrap-around. In this paper we describe a vector pointer setup by the tuple

$$(v, (\Delta_0, \Delta_1, \Delta_2), \delta, \rho),$$

which means that the indices of the vector pointer are initially set to $(v_0, v_1, v_2, v_3) =$

$$(v, v + \Delta_0, v + \Delta_0 + \Delta_1, v + \Delta_0 + \Delta_1 + \Delta_2),$$

and after post-increment the value of each v_i becomes

$$v_i - (v_i \bmod \rho) + ((v_i + \delta) \bmod \rho).$$

For example, a VP that toggles between accessing two consecutive quadruples of vector element registers starting at address v (divisible by 8) has the setup $(v, (1, 1, 1), 4, 8)$.

Additional instruction forms support more general updates of the VPR indices, and a VPR can also be explicitly modified by a special instruction. Note that in all cases

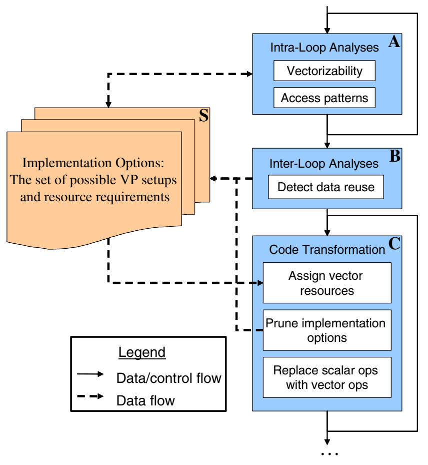


Figure 3: SIMdD Vectorization Framework

the index updating takes place in the Vector Pointer Unit (VPU), itself an SIMD functional unit, and the updated indices are written back into the same VPR.

3. THE ELITE COMPILER

The vectorization analyses and transformations described in this paper were implemented in a new "vectorizer" compilation pass that we added to an optimizing compiler. The compiler is based on Chameleon, an IBM VLIW Research Compiler [14]. With an enhanced form of Dependence Flow Graph (DFG) [20] and Static Single Assignment (SSA) (see, e.g. [16]), Chameleon has a repertoire of standard SSA-based optimizations and provides a rich infrastructure for compiler development. An outline of the vectorizer is given in Figure 3. Our vectorizer follows the loop-based vectorization approach because of our interest in exploiting reuse, and because it was appropriate for the kernels we focused on.

The first analysis phase (block A in Figure 3) identifies vectorizable loops using standard tests that include memory and data dependence analysis [23]. In addition, access patterns and memory alignment of data references are identified at this stage in order to compute the minimum set of resources required for vectorizing each loop, and also for eventual vector pointer setup. This information (represented by block S) defines a set of possible options for laying out data in the VEF (i.e. VEF allocations), and a set of options for accessing that data (i.e. VP setups). VP setups and VEF layouts are discussed in detail Section 5 and Section 6. These sets of implementation options may be augmented during cross-loop analysis (block B) as reuse opportunities are detected.

The vectorizer now decides upon a vectorization scheme (block C) guided by the analysis results summarized in block S. This implies

1. assigning operations to vector units (SIMdD/SIMpD), and

2. designating both VEF areas and VP setups for SIMdD assigned operations.

Operations are assigned to vector units according to the following criteria:

- Functionality constraints — if the necessary functionality is supported by only one of the units, then the operation is assigned to that unit.

- If there are opportunities for data reuse or needs for data reorganization, then the operation is assigned to the SIMdD unit.

- Otherwise latency, load balancing, and other factors are considered.

Once all operations have been assigned, the vectorizer selects for each memory reference that was mapped to the SIMdD unit a VEF assignment from the pool of available options (block S). It proceeds in a greedy manner, preferring the most profitable assignment for each array reference, pruning other options as resources become exhausted.

The final phase in the vectorizer performs the actual code transformation. This includes transforming the loops (unrolling, unroll-and-jam [3], variable expansion) and replacing scalar operations by vectorized counterparts, updating the DFG accordingly. Subsequent phases that take place after the vectorizer (such as the instruction scheduler) may apply further changes to the VPR setup in order to remove VEF anti-dependencies.

4. THE MAIN KERNELS

We now present two fundamental kernels in the multimedia domain: FIR filter and Viterbi decoder. We will use them in the following sections to demonstrate how various vectorization issues are handled when programming or compiling for a SIMdD architecture.

4.1 FIR Filter

The FIR filter, one of the most basic DSP kernels, performs filtering of speech signals in modern voice coders such as the ETSI GSM EFR/AMR or ITU G.729, and in many other signal processing areas. For a filter length M, coefficients array $h[0,...,M-1]$, input sequence $x()$ and output sequence $y()$, the FIR algorithm computes the following mathematical relationship among these signals in the time domain:

$$y(n) = \sum_{i=0}^{M-1} h(i)x(n-i).$$

Usually the output $y(n)$ is needed for several values of n, so several outputs may be computed in parallel. The number of outputs computed together is called "frame size" and is denoted by N.

The FIR kernel may be vectorized in two fashions: inner loop vectorization (by accumulator variable expansion), in which each iteration of the vectorized inner loop generates a single output (Figure 4a), or outer loop vectorization (by unroll and jam), in which several outputs are computed in parallel (Figure 4b).

4.2 Viterbi Decoder

One of the most important uses of the Viterbi Decoder algorithm is finding the most probable transmitted sequence of convolutional coded (CC) sequence. Such codes are widely used in digital communication. The most probable solution is also known as the Maximum-Likelihood (ML) solution.

The Viterbi Algorithm includes three steps. In the first step, branch metrics are computed. In the second step, the

```
for (int i = 0; i < N; i++) {
  ytemp(0,1,2,3) = {0,0,0,0};
  for (j = 0; j < M; j+=4) {
    ytemp(0,1,2,3) += h(j,j+1,j+2,j+3)
      * x(i-j+M-1,i-j+M,i-j+M+1,i-j+M+2);
  }
  y(i) = ytemp0 + ytemp1 + ytemp2 + ytemp3;
}
```

(a) Accumulator Variable Expansion

```
for (int i = 0; i < N; i+=4){
  y(i,i+1,i+2,i+3) = {0,0,0,0};
  for (j = 0; j < M; j++){
    y(i,i+1,i+2,i+3) += h(j)
      * x(i-j+M-1,i-j+M,i-j+M+1,i-j+M+2);
  }
}
```

(b) Unroll and Jam

Figure 4: Vector FIR Pseudo Code

```
for (j=0; j < numStates/2; j++) {
    y  = M[metric[j]];
    s0 = oldStates[2*j  ] + y;
    s1 = oldStates[2*j+1] - y;
    trace <<= 1;
    if (s1 < s0) {
      trace |= 0x1;
      s1    = s0;
    }
    newStates[j] = s1;
}
```

Figure 5: C Code for the Viterbi Add-Compare-Select Kernel

algorithm performs a maximization of the Likelihood function through a sequence of add-compare-select (ACS) operations, which computes an array of new state metrics based on the current state metrics and the set of the branch metrics computed in the first step. The corresponding ACS operations will be referred to as *butterfly operations.* In the third step, a traceback operation is performed from the end of the trellis, in which the most likely path through the trellis is identified and the data corresponding to the branches on that path is detected.

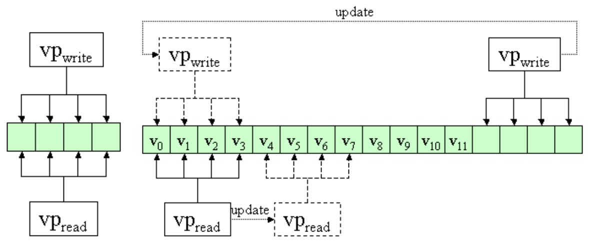
The parameters defining the Viterbi decoder complexity are the rate $1/n$ and the constraint length K. As typically $K = 9$ (e.g. in 3G cellular networks), the most computationally intensive part is the second (ACS) step, and we therefore concentrate on it.

The code in Figure 5 illustrates one half of the Viterbi Butterfly (the second half is similar): $s0$, $s1$ are taken from the *oldStates* array, and y is the relevant branch metric update taken from the branch metric array M. The decisions are packed into *trace*. The maximum is kept in the *newStates* array.

Note that the kernel can be written so that there are no cross-iteration data dependences, therefore the code is amenable to SIMD parallelism. Both the maximization and the trace update need special care, though.

5. DATA MANAGEMENT

Multimedia computations often deal with very long data streams. Therefore, questions like: "Does the data fit within the vector registers?", "How should the data be loaded into the vector registers?" and "Can the data be reused?" — are all critical to efficient programming in this domain. In this section we discuss two data management related programming techniques: register renaming and software caching. Both techniques are known and widely used, but are especially suitable to SIMdD where the VEF with its VPs supply a new and very powerful way of implementing these optimizations. These techniques were also automated and incorporated into the eLite compiler.



(a) Naive (b) With Pre-Loading

Figure 6: Vector Pointer Setup Example

5.1 Register Renaming and Rotating Registers

First let's examine how to program an inner-loop vectorized FIR kernel (Figure 4a). Assuming that the computation will be performed in the VEU unit, array h is to be written (loaded) into the VEF using vp_{write} and then manipulated using vp_{read}. A possible setup for these vector pointers is $vp_{read} = vp_{write} = (v, (1, 1, 1), 4, 4)$, in which case the vector pointers rotate over the same 4 elements, as depicted in Figure 6a.

Such a "Naive" data layout occupies minimal VEF space, but at the price of creating anti dependencies which severely constrain instruction scheduling and software pipelining. Increasing the number of elements over which the vector pointers rotate, to e.g. 16: $vp_{read} = vp_{write} = (v, (1, 1, 1), 4, 16)$, as shown in Figure 6b, will occupy more vector registers, but at the same time will increase the distance of these dependencies and enable aggressive code motion, including scheduling loads early in a software pipelined way to hide load latencies. In traditional architectures, register renaming poses a difficulty [16] as several values are kept "alive" at the same time, requiring more than one physical register and hence duplication of code or rotation of registers, either by software [3] or by special hardware [9].

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.

In the eLite DSP, hiding load latencies (and similarly more general software-pipelining optimizations) are performed by

1. placing loads in the loop prolog,
2. extending the VEF area reserved for the array so that it could contain all the loaded elements until they are last used, and
3. modifying the pointer setup accordingly.

In the example in Figure 6b, three loads can be hoisted to the loop prolog to pre-load twelve elements. Accordingly, the VEF allocation changes from four elements (as in Figure 6a) to 16 elements, so that there is enough space to hold data three iterations forward. Finally, the setup of the vector pointers used to read and write is modified so that the autoupdate, which advances them forward by four elements (as before), will rewind to the beginning at the boundary of 16 elements (instead of 4).

Note that changing the number of registers over which to rotate is accomplished by simply changing a parameter in vector pointer setup (in the loop prologue), rather than reassigning registers to all relevant instructions or making any other modifications within the loop. This is a very important advantage for a compiler, because 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.

5.2 Data Reuse and Cache Management in the VEF

In the specific case demonstrated above (simple access pattern, no data reorganization required), the advantage of SIMdD over SIMpD is mainly in the simplicity it offers to the compiler and programmer, relieving them from the need to worry about register renaming when considering issues like VEF space and load latencies. The effectiveness of rotating registers is even more evident if data reuse can be exploited.

We illustrate the potential of data reuse using the outer-loop-vectorized FIR example (Figure 4b). There, in each iteration i of the outer loop, the inner loop accesses $M+3$ elements of the x array residing in VEF entries $[i, i+1, \ldots, i+M+2]$. Figure 7 shows how in each outer-loop iteration this shaded interval shifts 4 elements to the right. Note that this kind of access pattern requires many shuffling instructions in SIMpD.

Figure 7 illustrates the overlap that exists between the regions of array x that are accessed in different iterations of the outer loop. A proper layout of the data in the VEF will allow to take advantage of this data reuse, and avoid reloading the same elements multiple times.

Exploiting reuse opportunities to reduce memory access latencies overhead is known to have dramatic effect on application performance. This is especially true for DSP and multimedia kernels, with their tight loops and well-structured array based computations. A classic solution to this problem is aggressive usage of large multi-levelled cache hierarchies, often accompanied with various loop transformations to enhance temporal and spatial reuse [23, 11], as well as various hardware mechanisms to augment the cache performance [18, 4, 21]. 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.

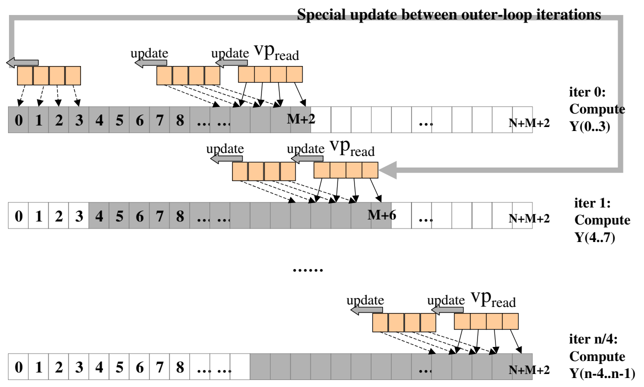

Special update between outer-loop iterations
update
update
vp$_{read}$
0 1 2 3 4 5 6 7 8 … … M+2 … N+M+2
iter 0: Compute Y(0..3)
M+6
iter 1: Compute Y(4..7)
......
iter n/4: Compute Y(n-4..n-1)

Figure 7: FIR with Flat Data Layout for Array x in VEF

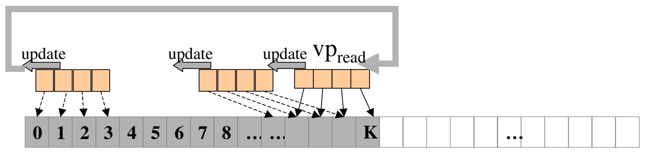


Figure 8: FIR with Cyclic Data Layout for Array x in VEF

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 that takes advantage of spatial and temporal reuse. 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 (such as previous iterations of a loop, as in the FIR example).

In the FIR example, one could place all $N+M+3$ elements of array x in the VEF, if there are available VERs, as depicted in Figure 7. However, this is not necessary — a more compact layout might achieve the same scheduling flexibility and reuse benefits, by streaming part of the data instead of pre-loading it. This is illustrated in Figure 8, where modulo arithmetic is used to wrap-around the same K VERs, where $K < N+M+3$. For example, if $K = M+6$, then after using elements 0,1,2,3 we replace them with elements $M+7, M+8, M+9, M+10$, and so on.

An implementation of the inner loop body is shown in Figure 9a. The loading of arrays x and h from memory into the VEF, as well as the setup code at the loop prolog are omitted for simplicity. Note that the code for the loop

```
vemul va1,(vp0),(vp1) || bnz          vemul va1,(vp0),(vp1) || vaadd va0,va0,va2 || bnz
nop                                   vemul va2,(vp0),(vp1) || vaadd va0,va0,va3
nop                                   vemul va3,(vp0),(vp1) || vaadd va0,va0,va4
vaadd va0,va0,va1                     vemul va4,(vp0),(vp1) || vaadd va0,va0,va1
```

(a) Naive **(b) Software Pipelined**

Figure 9: FIR Inner-Loop Implementation

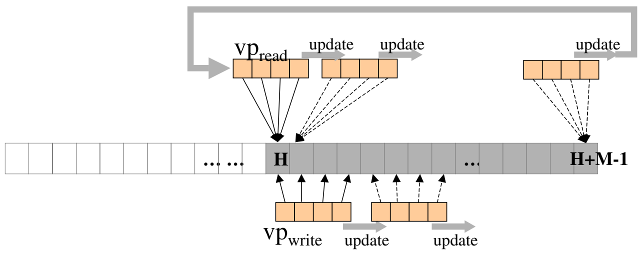


Figure 10: VP Setup for the Coefficients Array h in FIR

body remains the same regardless of which of the two data layouts (flat or cyclic) is used. Only the setup code at the loop prologue is affected. For the "flat" data layout, the vector pointer that is used to access array x in the VEF is setup as

$$vp = (M-1, (1,1,1), -1, \infty),$$

whereas for the wrap-around case the setup is

$$vp = (M-1, (1,1,1), -1, mod).$$

Each time a VEU multiplication uses vp, the autoupdate mechanism associated with all VEU operations moves the pointer to its next position. In the "flat" case, an additional special update instruction will have to be used between outer loop iterations to bump vp to its next starting position.

Figure 10 illustrates the setup of vector pointers used to access the coefficients array h. The register used to read the array in the VEF is setup as

$$vp_{read} = (H, (0,0,0), 1, M),$$

and the array is written into VEF using

$$vp_{write} = (H, (1,1,1), 4, \infty).$$

To avoid idling while waiting for the result of the multiplication, the loop is software pipelined (see Figure 9b). We have to perform register renaming and use different names for the result registers ($va1, \ldots, va4$), but the same two vector pointer registers are used for all multiplications.

In the case of the FIR kernel the compiler is able to match the programmer performance-wise, due to its ability to expose these reuse opportunities; Its inter-loop analyzer propagates memory references information through the loop hierarchy tree in a leaf-to-root manner, similarly to [5]. When detecting overlapping accesses, the vectorizer can allocate VEF entries to support data sharing across loop iterations and between loops according to available resources.

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

6. EFFECTIVE DATA REORGANIZATION

Data reorganization problems 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 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.

The SIMpD architectures provide capabilities to pack and unpack data to and from vector registers, either during memory operations [1, 6], or within register files [1, 8, 19, 17, 22]. This approach for reorganizing vector data has a number of drawbacks. Reordering within memory operations increases the already long latency of memory operations, may cause a series of cache misses when addressing several remote locations and misses spatial reuse opportunities [23]. Special permutation instructions, on the other hand, introduce additional overheads. When reordering is applied repeatedly (e.g., within a loop), such overheads are incurred each time.

SIMdD allows permuting the access without actually moving any data. Multiple instructions that need the same permutation may (re)use the same VP thanks to its implicit update capabilities. The penalty at each permuted access is thus replaced by a one-time penalty of setting up a VP. We start with a description of a simple special case of reorganization: alignment in memory. We then describe how to handle interleaved access and present the general alternatives and considerations in setting of data reorganization in SIMdD.

6.1 Alignment in Memory

Memory alignment constraints raise problems that can be handled using data reordering mechanisms. 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. The memory architecture in the eLite DSP, for example, restricts vector data accesses to 4 consecutive 16-bit elements in memory aligned on 64-bit boundary, with an option to disable storing any subset of elements. Techniques used to avoid these penalties such as loop-peeling [2] or dynamic alignment detection [12] are not always applicable and increase code size. Techniques that try to confront this problem usually incur a penalty that grows linearly with the data set size [13, 6].

Memory alignment problems can be treated as a special case of data reordering, where the access to a contiguous data set is slightly shifted to comply with the memory alignment constraints. In order to read from an array that is not aligned in memory using SIMdD, the array can be loaded into the VEF as if it starts at the nearest aligned address preceeding the first element, and ends at the nearest aligned address following the last element. This is accomplished by having one extra vector load instruction (that brings data from memory and places it in the VEF) in the loop prologue; it requires a few extra spaces in the VEF, but the vector pointer pattern remains the same for all loads. The vector pointers used to read the loaded data from the VEF also retain the same pattern, but skip over the first few elements. Storing data to unaligned memory locations is handled similarly to loads, except that the first and last vector store instructions are masked appropriately so as not to write past the bounds of the target array. We are thus able to load and store arrays into unaligned memory locations using accesses to aligned memory only, with only a small constant overhead.

To illustrate the solution for misaligned memory access, consider an FIR where the h array may not be properly aligned in memory. Assume that the array h is stored in memory starting at address s and ending at address t, where addresses are expressed in 16-bit units (so $t - s = |h| - 1$). Let $b = s \bmod 4$ and $e = t \bmod 4$. If $b \neq 0$ or $e \neq 3$ we cannot read h exactly from memory into the VEF, because h is not properly aligned. The solution is to load a properly aligned interval containing h, starting from $s-b$ until $t+3-e$. This may load at most six extra elements, whatever the length of array h. We could use, for example,

$$vp_{write} = (A, (1, 1, 1), 4, \infty)$$

for writing into the VEF and

$$vp_{read} = (A + b, (1, 1, 1), 4, \infty),$$

for reading from the VEF, thus skipping the extra b elements at the beginning.

6.2 Interleaved Access Pattern

Reordering of data is required when an access pattern is not consecutive. The constant-stride pattern (d, d, d) is probably the most widespread non-consecutive pattern in the DSP context. With $d = 2$ this pattern appears in computations on complex numbers, where the real and imaginary parts are interleaved in the same input or output array. Complex inputs must be de-interleaved to carry out the SIMD computations, and complex outputs need to be re-interleaved. Decoders and encoders for interleaved codes and computations on very long data types also give rise to this pattern.

Consider, for example, the complex FIR filter:

$$\begin{aligned}\mathrm{Re}(y_n) &= \sum_{i=0}^{M-1} [\mathrm{Re}(h_i)\mathrm{Re}(x_{n-i}) - \mathrm{Im}(h_i)\mathrm{Im}(x_{n-i})],\\ \mathrm{Im}(y_n) &= \sum_{i=0}^{M-1} [\mathrm{Re}(h_i)\mathrm{Im}(x_{n-i}) + \mathrm{Im}(h_i)\mathrm{Re}(x_{n-i})].\end{aligned}$$

The pseudo-code is given in Figure 11, under a fairly realistic assumption that arrays h and x contain the real and imaginary elements interleaved. In eLite's SIMdD architecture the data does not need to be moved within registers to access an even tuple or an odd tuple. By setting

$$vp_{write} = (v, (1, 1, 1), 4, \infty)$$

$$vp_{readEven} = (v, (2, 2, 2), 8, \infty)$$

$$vp_{readOdd} = (v + 1, (2, 2, 2), 8, \infty)$$

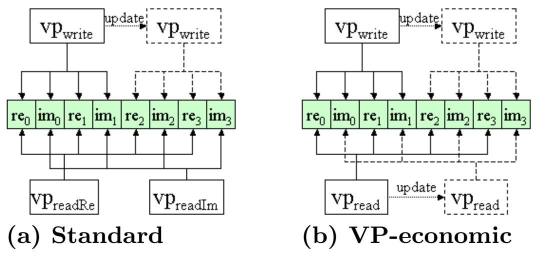


Figure 12: Reordering at VEF Read

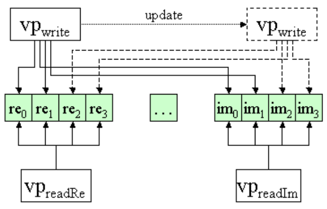


Figure 13: Reordering at VEF Write

we achieve the required reorganization on-the-fly. Accordingly, a straightforward setup of VPs for the multiplication instructions uses four pointers, one for each real or imaginary part of each input array, using the following settings: $(v1, (2, 2, 2), 8, 8)$, $(v1{+}1, (2, 2, 2), 8, 8)$ and $(v2, (2, 2, 2), 8, 8)$, $(v2 + 1, (2, 2, 2), 8, 8)$. Figure 12a depicts this setup option for one array. In every iteration each pointer is used once without update and once with implicit update by 8.

6.3 VP Setup Alternatives for Data Reordering

In the previous subsections we described how the VEF can serve as a "reorder buffer" for data reordering. We conclude this section by presenting additional alternatives for vector pointer setup and VEF allocation, and point out the considerations that determine which vector setup will be used in each case.

Recall that 4 VPs were used for the multiplication instructions in the complex FIR example. Because these VPs have the same pattern $(2, 2, 2)$, more VPs (if available) can be used to eliminate dependencies and produce more regular code: each VP can be split into two VPs, doubling the implicit update. This is similar in concept to induction variable expansion. The number of VPs can also be reduced, using a smaller implicit update, but at the cost of an additional explicit update. In the example, each iteration can use a single vp_{read} (instead of $vp_{readEven}$ and $vp_{readOdd}$) set to $(v1, (2, 2, 2), 1, \infty)$, and an explicit update of 7 in every iteration (see Figure 12b vs. Figure 12a). These alternatives demonstrate a tradeoff between the number of VPs that are used and the number of instructions that are required to implement the access pattern.

```
    for (int i = 0; i < N; i+=8){
        y(i,i+2,i+4,i+6) = {0,0,0,0};
        y(i+1,i+3,i+5,i+7) = {0,0,0,0};
        for (j = 0; j < M; j+=2){
            //every tuple accesses either only even or only odd elements
            int p = i-j+2M;
            y(i,i+2,i+4,i+6) += h(j)*x(p-2, p, p+2, p+4) - h(j+1)*x(p-1, p+1, p+3, p+5);
            y(i+1,i+3,i+5,i+7) += h(j)*x(p-1, p+1, p+3, p+5) + h(j+1)*x(p-2, p, p+2, p+4);
        }
    }
```

Figure 11: Pseudo-Code for an Unroll-and-Jam Vectorization of Complex FIR

So far we have described how a non-consecutive data access can be handled by placing the data consecutively in the VEF and reading it using VPs which are set up with non-consecutive access patterns. We refer to these setup schemes as "reorder-on-read". A symmetric "reorder-on-write" alternative is to place the data non-consecutively into the VEF and read it consecutively. Referring to Figure 13, the pattern of the vector pointer VP_{write} used by the load is $(v0, (\delta, 1-\delta, \delta), 2, 8)$, where $\delta = v1 - v0$. In the general cases of a (d, d, d) pattern with d dividing the vector length 4, reorder-on-write requires a VEF allocation of d areas of size 4 each, rather than a single area of size $4d$ as required by reorder-on-read. Such a disjoint allocation might be preferred, depending on VEF availability and wrap-around mechanism restrictions.

The complex FIR 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 its elements along rows and along columns. Vector pointers can be used to implement such multiple accesses efficiently, again without reordering the data itself.

The eLite compiler considers these VP setup alternatives as part of its vectorization framework.

7. RANDOM ACCESS PATTERN: THE VITERBI DECODER

In this section we show how SIMdD supports complicated access patterns found in the Viterbi algorithm, including an interleaved pattern and an arbitrary pattern. The source code for the Viterbi butterfly kernel appears in Figure 5 and the assembly code is shown in Figure 14. The M array containing the metric update is pre-computed and stored in the VEF. The size of the M array is 2^n, where typical values for n (such as used in 3G cellular networks) are 2 or 3. The *metric* array which is kept in memory is constant, and its values depend on the convolutional encoder parameters. It points to elements in the M array, which in our implementation correspond to locations in the VEF. Values of consecutive indices in the metric array are typically not contiguous, so the selection of the correct element in SIMD manner is awkward on SIMpD architectures, but fits perfectly into our SIMdD architecture. Such random access patterns emphasize the ability of SIMdD architecture to efficiently vectorize kernels that would otherwise require many data shuffling instructions.

```
                           ;a4 points to metric
ldvpu vp0,8(a4)            ;vp0=*a4++
veadd va0,(vp4)u,(vp0)u    ;va0=old[2j']+y,j'++
vesub va1,(vp5)u,(vp0)u    ;va1=old[2j+1]-y,j++
vamax va1,va0,vm0          ;va1=max(va0,va1),
                           ;vm0=(va0 >= va1)
mval (vp6)u,va1            ;new[j"]=va1,j"++
```

Figure 14: Assembly Code for the Viterbi Add-Compare-Select Kernel

Both *oldStates* and *newStates* arrays are kept in the VEF to reduce the amount of memory transfers. Vector pointer $vp0$ points to y, and is set up by loading *metric* from memory while disabling autoupdate because it is used twice (see `ldvpu` instruction in Figure 14); $vp4$ is set to point to even samples in *oldStates* array:

$$vp4 = (\&oldStates[0], (2, 2, 2), 8, \infty).$$

Similarly, $vp5$ is set to point to the odd samples:

$$vp5 = (\&oldStates[1], (2, 2, 2), 8, \infty).$$

Vector accumulator $va0$ holds $s0$ for 4 iterations of the scalar loop (scalar expansion), and similarly $va1$ for $s1$. $vp6$ points to samples in *newStates* array so it is set to

$$vp6 = (\&newStates[0], (1, 1, 1), 4, \infty).$$

The implementation of the trace shift register is also very important. The Compare-Select is one of the essential operations in the Viterbi kernel, so a specialized max instruction is available: in addition to selecting the maximum value, it also keeps track which of the two operands is larger by setting a bit in a Vector Mask (VM) register. The shift register is implemented by (unrolling and) register renaming the VM register elements. Having 8 VM registers of size 4 bits each, 8 butterflies halves are repeated before the entire VM register is stored at once. By this method no shift instructions are performed on the trace register.

Aggressive unrolling of the loop is applied in order to keep the pipeline full. The code achieves asymptotic efficiency of 1 cycle per butterfly, which is the best one can achieve with 4-way SIMD. The memory traffic is kept minimal because only the metric array is loaded from memory while the states are kept in the VEF.

The needed VEF size includes array M (8 elements for $n = 3$) and the states arrays (512 elements for $K = 9$). Memories tend to come in radix-2 sizes, so it is preferred to squeeze the kernel into VEF of size 512. This can be achieved by overlaying the last 8 elements of *newStates* with the M array, and carefully pipelining the epilog of the kernel.

While the compiler is able to auto-vectorize the viterbi kernel, it is not able to achieve the same efficiency as the assembly programmer. The main difficulty for the compiler in this case is memory disambiguation of the *oldStates* and *newStates* arrays. Recall that the *newStates* array in one invocation of the ACS computation serves as the *oldStates* array for the next invocation of the ACS computation. This is achieved in the C code using pointer swapping, which currently does not allow the compiler to prove that the two pointers are anti-aliased in each ACS invocation. Therefore, the compiler is not able to exploit the data reuse between subsequent ACS computations, and much more memory traffic is generated as the *oldStates* and *newStates* arrays are entirely loaded and stored from/to memory each time. However, the body of each ACS computation is vectorized by the compiler similarly to the scheme described above.

8. EVALUATION

8.1 SIMdD vs. SIMpD - Pros and Cons

The advantages of the SIMdD architecture are not effective in benchmarks where data is already aligned, contiguous and not reused. For example, a kernel that adds together aligned vectors from memory is perfectly amenable to regular SIMpD architectures. In such benchmarks our results for using SIMdD are slightly worse than using SIMpD, because the former requires more registers to be set in the prolog sections. This degradation is insignificant in real applications, because such "summation" kernels are of linear complexity where performance bottlenecks are usually attributed to sections of higher complexity, and because prolog code may be scheduled early (in former kernel epilog) thereby hiding its latencies.

The SIMdD architecture shines on kernels where the disjoint capability and data reuse are really required. Such kernels include, for example, the Viterbi Decoder and parametric data reorder such as ZigZag scan used in many video codecs. Modern SIMpD processors can cope reasonably well with stride 2 access patterns (that appear e.g. in complex arithmetic), but have serious difficulties dealing with arbitrary strides. SIMdD architecture can achieve full asymptotic efficiency even in such benchmarks — the appropriate execution units are always working, after a short prolog. In addition to achieving optimal performance efficiency, the large amount of data reuse achieved for such kernels reduces the memory accesses significantly, thus reducing the needed power. Beyond the power and performance advantages, such kernels also reflect the programming simplicity that a SIMdD architecture facilitates.

We are unable to present a quantitative comparison between SIMdD and SIMpD techniques for cases where data reorganization is needed, since no SIMpD reorganization mechanisms (such as permute instructions) are available in eLite. On the other hand, conducting a meaningful comparison between eLite's SIMdD architecture and some other SIMpD architecture is problematic, because it will not reflect the tradeoffs between SIMpD and SIMdD alone, but rather the effect of a mixture of parameters. It is beyond the scope of this paper to quantitatively compare the two programming models; We are rather interested in understanding the qualitative differences between the two models and consequently the new challenges and opportunities that SIMdD suggests for SIMD programmers and vectorizing compilers.

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

8.2 Experimental Evaluation of the Compiler

In this section we compare the performance of code generated by the eLite compiler with code optimized for the eLite architecture independently by expert assembly programmers, the latter achieving asymptotic efficiency. The experimental results were generated automatically using a cycle-accurate eLite simulator and profiler.

Table 1 provides a brief description of the benchmarks we used.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), and are representative of the main computation kernels in our target application domain.

Figure 15 displays the relative execution time of compiled codes, normalized to the execution time of hand-optimized codes. Both the hand-written and the compiler-generated codes are vectorized using the VEF and the techniques described in this paper. The figure displays two results for the compiler; one, denoted "data fits in VEF" was achieved under the assumption that the data fits in the VEF, a reasonable assumption in the respective application domain. The other, "general data size" is without this assumption. For benchmarks originally tailored to a fixed data size we do not present the "general data size" result.

For the general data size case, the average increase in execution time of compiled vs. hand-optimized codes is 35%, with an average of 25% difference for single-nested loop kernels and an average of 44% for multi-nested loop kernels. This is primarily due to 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 scheduling is more efficient: the average performance difference is 15%, with an average of 7.5% for single-nested and 19% for multi-nested loop kernels.

9. CONCLUSIONS

This paper presents a set of vectorization techniques for an SIMdD architecture. We show how the novel capabilities of the architecture can provide low-overhead and efficient

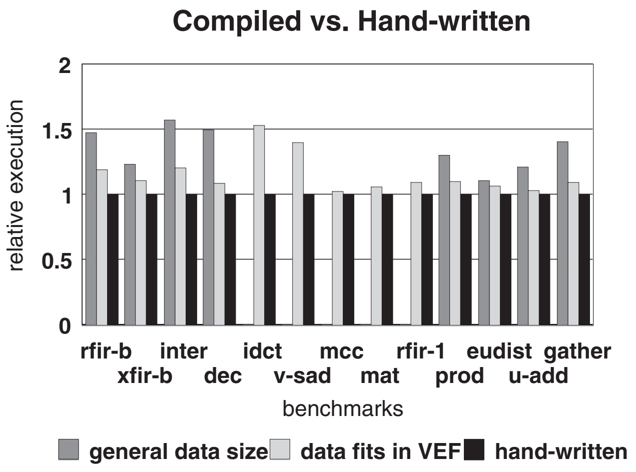


Figure 15: Experimental Results

solutions to the traditionally difficult problems of data reordering, data misalignment, and register renaming. We also applied these techniques to compiler vectorization and data reuse optimizations. We demonstrate that the performance of the code generated by a compiler using these techniques is comparable to hand-optimized code for a set of "regular" benchmarks representative of the DSP domain, with overhead below 20%.

By combining novel architecture capabilities with innovative vectorization techniques we showed how data reorganization problems can be solved seamlessly and effectively, opening new opportunities for dealing with other critical issues of data reuse and efficient scheduling in the context of SIMD vectorization. This is another major step towards effective use of SIMD architectures in the DSP domain.

10. ACKNOWLEDGMENTS

We would like to thank Amir Geva for his eLite tools support, Krishnan Kailas for his contributions to the eLite compiler, Uzi Shvadron, Jaime Moreno and the world-wide eLite team for continued discussions.

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

[4] William 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] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimizations. In *Ninth Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[6] Jesus Corbal, Roger Espasa, and Mateo Valero. Exploiting a new level of DLP in multimedia applications. In *Intl. Symposium on Microarchitecture*, pages 72–, 1999.

[7] Paul D'Arcy and Scott Beach. StarCore SC140: A new DSP architecture for portable devices. In *Wireless Symposium*. Motorola, September 1999.

[8] K. Diefendorff and P. K. Dubey et al. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, March-April 2000.

[9] Gautam Dohsi, Rakesh Krishnaiyer, and Kalyan Muthukumar. Optimizing software data prefetches with rotating registers. In *PACT*, pages 257–267, 2001.

[10] Texas Instruments. www.ti.com/sc/c6x, 2000.

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

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

[13] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Memo 621, MIT LCS, November 2001.

[14] 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.

[15] Jaime 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. *IBM Journal of Research and Development*, March 2003.

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

[17] Huy Nguyen and Lizy Kurian John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *Intl. Conf. on Supercomputing*, pages 11–20, 1999.

[18] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design and Test Conf.*, March 1997.

[19] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, pages 43–45, August 1996.

[20] 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.

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

[22] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *PACT*, 2002.

[23] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.