An Integrated Simdization Framework Using Virtual Vectors

Peng Wu Alexandre E. Eichenberger Amy Wang Peng Zhao

IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{pengwu,alexe}@us.ibm.com

IBM Toronto Laboratory
Markham, Ontario, Canada
{aktwang,pengz}@ca.ibm.com

ABSTRACT

Automatic simdization for multimedia extensions faces several new challenges that are not present in traditional vectorization. Some of the new issues are due to the more restrictive SIMD architectures designed for multimedia extensions. Among them are alignment constraints, lack of memory gather and scatter support, and the short and fixed-length nature of SIMD vectors. Since these constraints affect some very basic components of a program, a compiler must not only provide solid solutions to individual issues, but also take an integrated approach to address these constraints in combination.

In this paper, we propose a simdization framework that addresses several orthogonal aspects of simdization, such as alignment handling, simdization of loops with mixed data lengths, and SIMD parallelism extraction from different program scopes (from basic blocks to inner loops). The novelty of this framework is its ability to facilitate interactions between different techniques based on the simple intermediate representation of virtual vectors. Measurements on a PPC970 with a VMX SIMD unit indicate speedup factors of up to 8.11 for numerical/video/communication kernels and speedup factors of up to 2.16 for benchmarks, when automatic simdization is turned on.

1. INTRODUCTION

Most modern microprocessors have adopted multimedia extensions to achieve higher performance on increasingly important multimedia workloads. Despite the variety of multimedia extensions on the market, at the core, these extensions can be characterized as **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) units operating on densely-packed, short vectors. Examples of such systems are VMX for PowerPC [1], SSE for Pentium [2], and the CELL processor [3].

While SIMD units are becoming ubiquitous, their impact on mainstream program performance is still under their full potential. Programming SIMD units by hand is both time consuming and error prone. In the meantime, optimizing compilers still have difficulties in generating efficient codes for SIMD units, despite considerable expertise in compiling for vector supercomputers. In this paper, we refer to SIMD vectorization as **simdization**.

The underlying reasons are two folded. On the one hand, many issues faced by automatic exploitation of SIMD parallelism are caused by the new (sometimes more restrictive) hardware features in today's SIMD architectures.

- Most SIMD units, such as VMX and CELL, only access chunks of *contiguous* memory that are *aligned* at a multiple of the vector register length. The constraints on memory alignment and contiguity result in simpler, faster memory operations as data do not span multiple cache lines and require no hardware shuffling network between the memory and SIMD units. In contrast, vector machines such as Cray T90 can load memory vectors using a vector of random addresses regardless of memory alignment. Instead, the burden to gather, scatter, or re-align data is shifted to the software.
- SIMD units often perform operations on vectors of various data element lengths. For example, VMX supports vector operations on 4 int/float, 8 short, or 16 char data packed into a single 16-byte vector. One implication of the packed representation to the compiler is that the number of vectors involved in a computation changes during data conversions of different data lengths. For example, during a conversion from a 1-byte char to 2-byte short, a packed vector of short should expand into two vectors of int.
- Because SIMD units are in general more constrained than the scalar units available on the same machine and because of the short nature of SIMD vectors, the interaction between simdization and other compiler optimizations has significant impact to simdization performance. One such example is the interaction between simdization and exploiting instruction level parallelism. Without extra care, the compiler can easily simdize a loop while actually degradating the performance.

On the other hand, the applications targeted by SIMD units are often more complex than those targeted by traditional vector machines. The new application domain brings the compiler both challenges and opportunities.

- While long running loops are the primary source of traditional vector parallelism, the short nature of SIMD

ICS'05, June 20-22, Boston, MA, USA.

```
short input[], coef[];
for (i=0; i<NInputs; i++) {
  int sum = 0;
  for (j=0; j<16; j++)
    sum += input[k+j] * coef[j];
  output[i] = sum;
}
```

Figure 1: A typical FIR loop.

vectors greatly increases its applicability for short, unstructured computations. In many multimedia codes, SIMD parallelism may be found in computation within a basic-block, in an innermost loop (sometimes with very short trip count), or even in an innermost loop nest. This presents the compiler the opportunity to jointly extract SIMD parallelism from various program scopes.

- While loops with uniform data type are dominant in traditional vector parallelism, the type mix found in multimedia applications is extremely rich and data conversions are frequent. For example, data are often stored as 1-byte char and operated on as 2-byte short. Without extra care, one may easily utilize the full computation bandwidth of the machine, *e.g.*, computing 8 shorts at a time, while wasting half of the memory bandwidth, *e.g.*, by storing only 8 chars at a time. Thus, a compiler must be able to efficiently handle mixed data types in a loop to maximize both computation and memory bandwidth of SIMD units.

While several recent researches [4, 5, 6, 7] have focused on addressing individual simdization issues mentioned above, we also observe that the complexity of simdization is further compounded by the fact that these issues frequently occur all at once in real programs. Consider, for example, the Finite Impulse Response (FIR) computation shown in Figure 1. This code[1] exhibits several simdization issues we mentioned, such as misaligned accesses, data conversion, and SIMD parallelism across nested loops.

In this paper, we propose a simdization framework that integrates different aspects of simdization and enables the interoperability among them. One key result is that we can now (jointly) extract SIMD parallelism from multiple program scopes – basic-block, innermost loops, and innermost loop nests – using for each scope a different technique that is best suited for its individual challenges. In addition, the extraction of SIMD parallelism is also fully integrated with two other orthogonal aspects of simdization, alignment handling and data conversion, so that the compiler may efficiently simdize codes involving arbitrary combinations of misalignment and data type mix in loops with any of three levels of SIMD parallelism.

The interoperability of the simdization framework is achieved by the use of virtual vectors and a multi-step devirtualization process. A virtual vector can have properties that are less constrained than those of hardware physical vectors, such as having arbitrary length or no alignment constraints. This enables some of the early transformations, such as extracting SIMD parallelism from basic blocks and loops, to operate more easily because they are free of many architectural constraints of SIMD units. Virtual vectors are then gradually devirtualized to physical vectors where actual architecture constraints, such as memory alignment and physical vector length and operations, are handled.

Based on this framework, we propose four techniques that jointly exercise many (orthogonal) aspects of simdization:

- simdization of loops with basic-block and loop-level parallelism,
- simdization of loops with data conversion,
- simdization across loop nests (with short inner-loops),
- simdization resulting in mixed scalar and SIMD codes.

Measurements on a PPC970 with a VMX SIMD unit indicate speedup factors of up to 8.11 (2.63 harmonic mean speedup factor) for 6 kernels extracted from numerical, video, and communication applications. Measurements also indicate speedup factors of up to 2.16 (1.20 harmonic mean speedup factor) for 9 benchmark programs. Factors are the ratio of the times for a non-simdized application over the simdized application achieved the same optimization level.

The rest of the paper is organized as follows. Section 2 introduces the overall simdization framework. We then describe four simdization techniques based on virtual vectors in Section 3, 4, 5, and 6, respectively. Experimental results are presented in Section 7. Section 8 discusses the related work and we conclude in Section 9.

2. OVERVIEW OF FRAMEWORK

In this section, we introduce virtual vectors and give an overview of the proposed simdization framework.

2.1 Virtual Vectors

Virtual vectors are an abstraction above the physical vectors provided by a hardware SIMD unit. Specifically, a virtual vector has an arbitrary data length and no alignment constraint. It is represented here by a ***V(n, t)*** tuple to indicate a vector of n packed elements of data type t. The number of element per vector, n, is a compile-time constant. The type of the vector elements, t, is either a primitive data type or is itself a vector.

The first property of a virtual vector is its byte length, $\texttt{len}(V(n,t))$, which is defined as $n * \texttt{len}(t)$. When t is primitive data type, such as `char` or `float`, $\texttt{len}(t)$ is simply the byte size of that type, *e.g.*, $\texttt{sizeof}(t)$. When t is a vector type, the length formula is recursively applied until a primitive type is reached.

This length property is not to be confused with the byte length of the physical vectors supported by the hardware, which is fixed and denoted here as P_{VL}.

The second property of a virtual vector is its alignment property. An *aligned* vector load/store abstracts the behavior of an aligned memory instruction. For example, a load of an aligned vector $V(n,t)$ from address *addr* loads $\texttt{len}(V(n,t))$ bytes from $\lfloor addr/P_{VL} \rfloor * P_{VL}$; whereas a load of an *unaligned* vector loads vector length bytes from an arbitrary address *addr*.

To conclude, we say that a virtual vector represents a physical vector if the vector is *aligned* and $\texttt{len}(V(n,t)) = P_{VL}$. For the rest of the paper, we assume that P_{VL} is 16 bytes.

[1]Some FIR implementations also completely unroll the innermost loop.

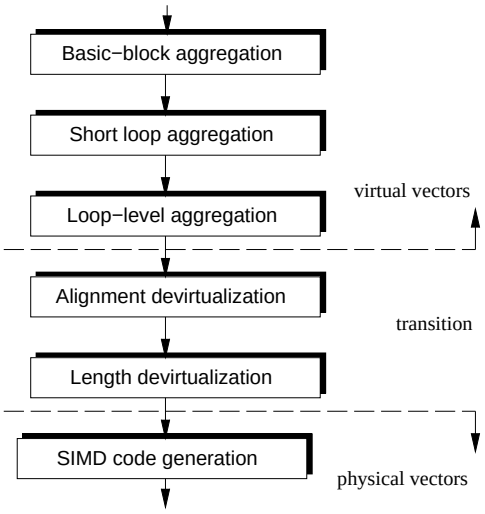

Figure 2: A simdization framework based on virtual vectors

2.2 Simdization Framework

In our framework, there are the six main transformation components, as depicted in Figure 2. The first three phases extract SIMD parallelism at different program scopes into generic operations on virtual vectors. The next two phases progressively devirtualize virtual vectors to match the precise architecture constraints. The final phase lowers the generic vector operations to platform specific instructions.

Phase I: Basic-block level aggregation. This phase extracts SIMD parallelism within a basic block by packing isomorphic computation on adjacent memory accesses to vector operations.

Vectors produced by this phase have arbitrary lengths and may be unaligned.

Phase II: Short loop aggregation. This phase eliminates simdizable inner loops with short, compile-time trip counts by aggregating static computation on stride-one accesses across the entire loop into operations to longer vectors. Given a short loop with compile-time trip count u, any data of type t in the loop becomes vector $V(u,t)$ after the aggregation.

Vectors produced by this phase have arbitrary lengths and may be unaligned.

Phase III: Loop-level aggregation. This phase extracts SIMD parallelism across loop iterations. Computations on stride-one accesses across iterations are aggregated into vector operations by blocking the loop by a factor of B. Any data of type t in the loop becomes vector $V(B,t)$ after the aggregation.

The blocking factor B is determined such that the byte length of each vector $V(B,t)$ is always a multiple of P_{VL} bytes, namely $B*\texttt{len}(t) \equiv_{P_{VL}} 0$. The smallest such blocking factor is

$$B = \frac{P_{VL}}{\text{GCD}(P_{VL}, \texttt{len}(t_1), \ldots, \texttt{len}(t_k))}, \qquad (1)$$

where GCD computes the greatest common divisor among all the inputs.

After the completion of this phase, vectors have byte lengths that are multiples of P_{VL} bytes but may still be unaligned.

Phase IV: Loop-level alignment devirtualization. This phase transforms loads and stores from possibly unaligned vectors to aligned vectors. We have devised an algorithm that can handle loops with arbitrary misalignments by realigning data in registers, while minimizing both memory and data reorganization overhead involved in data realignment [6].

In our algorithm, stride-one memory accesses across iterations are viewed as *streams*, where the alignment of the beginning of a stream is called *stream offset*. Two streams are considered relatively misaligned if they have different stream offsets. In this case, to ensure the correctness of the computation after simdization, the compiler inserts a pseudo operation called *stream shift* to shift in registers one stream to match the offset of the other stream. Stream shift is later mapped to a sequence of native data reorganization instructions.

At this stage, vectors are aligned and have byte lengths that are multiples of P_{VL} bytes.

Phase V: Length devirtualization. In this phase, vectors are first flattened to vectors of primitive types. The compiler then maps operations on virtual vectors to operations on multiple physical vectors. Sometimes, operations virtual vectors may be devirtualized to scalar operations when, for instance, the target platform does not support a particular SIMD operation. The decision is based on the length of the vector, whether the vector is aligned, and other heuristics that determine whether to perform the computation in vectors or scalars.

Vectors produced by this phase are now physical vectors.

Phase VI: SIMD code generation. This phase maps generic operations on physical vectors to one or more SIMD instructions or intrinsics, or to library calls according to the target platform.

In this framework, SIMD aggregation occurs within virtual vectors, which offer the compiler a useful abstraction close to vectors provided by traditional vector machines. This aggregation can solely focus on extracting SIMD parallelism from the more diverse sources present in modern multimedia applications. The logistic of dealing with reduced hardware support is then relegated to the later simdization phases that incrementally address issues such as alignment, short vectors, and packed data types. Finally, the flexibility of devirtualizing vectors to scalars makes it possible to revert some of the less optimal decisions made during early phases of simdization, when later a more global view of resources becomes available.

In the next sections, we will describe four techniques incorporated in this framework that clearly demonstrate the benefit of using virtual vectors and devirtualization.

3. SIMDIZATION WITH MULTIPLE SOURCES OF SIMD PARALLELISM

We describe in this section our integrated approach to extracting SIMD parallelism present in basic block and across consecutive loop iterations.

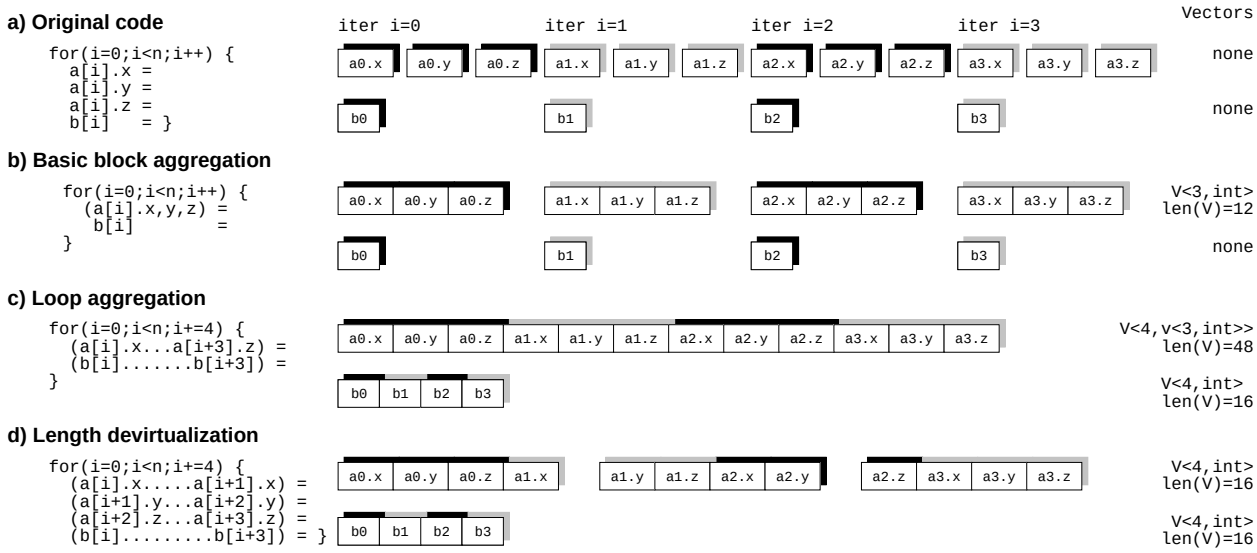

Figure 3: Mixed stride-one and adjacent accesses.

3.1 Mixed Sources of SIMD Parallelism

In the absence of hardware support for gather and scatter memory operations, the most cost-effective extraction of SIMD parallelism is to aggregate computation on a contiguous chunk of memory into vector operations. There are generally two instances in which contiguous memory locations are found in programs.

- At the basic-block level, multiple *adjacent memory accesses*, such as `a.x` and `a.y` or `b[i]` and `b[i+1]`, can be combined into an access to a contiguous chunk of memory.
- At the loop level, instances of a *stride-one access*, such as `c[i]`, across consecutive iterations are combined into an access to a contiguous chunk of memory.

These two types of accesses are often not compatible. For example, `d[i].x` and `d[i].y` may be adjacent memory accesses at the basic-block level, but are definitely non stride-one memory accesses, individually, at the loop level.

3.2 Integrated Loop Simdizer

Let us illustrate the combined extraction of SIMD parallelism within a basic block and among consecutive loop iterations using the example shown in Figure 3. We will follow this example through the relevant phases described in Section 2.2. While the actual algorithm takes all memory references into account, we focus here only on the stores generated by each of the loop's four statements, for conciseness. Each individual 4-byte store is represented by a box in Figure 3; the boxes' shadows are alternatively gray or black to distinguish consecutive iterations of the original loop.

Basic-block level aggregation. The compiler recognizes that the first three stores in Figure 3a, `a[i].x`, `a[i].y`, and `a[i].z`, are adjacent in memory. Assuming here that the right hand sides of the three statements are isomorphic, it aggregates the three statements into a vector of 3 integers stored into `a[i].x,y,z`, as shown in Figure 3b. The `b[i]` statement remain unchanged.

Loop-level aggregation. Recognizing the vector store `a[i].x,y,z` and the element store `b[i]` as stride-one accesses, it further aggregates these accesses across loop iterations. In doing so, it treats the new vector `a[i].x,y,z` statement no differently than any other statements in the loop. The only difference between the `a[i].x,y,z` and `b[i]` statements is that the first one generates a 12 byte value whereas the second one generates a 4 byte value.

During this phase, we extract SIMD parallelism among the smallest number of consecutive iterations while ensuring that each vector in the loop has a length that is a multiple of the physical vector length, 16 bytes here. The formula of Equation (1) determines that the optimal blocking factor is 4, since it aggregates 4 of the 12-byte `a[i].x,y,z` vectors into a new compound vector of 48 bytes and 4 of the 4-bytes `b[i]` values into a new vector of 16 bytes.

The resulting loop is shown in Figure 3c. Note that we may flatten vectors of vectors at any times, *e.g.*, simplifying $V(4, V(3, \texttt{int}))$ in the code above to $V(12, \texttt{int})$.

Alignment devirtualization. This step is necessary when array `a` or `b` are not aligned. Since both `a[i].x,...,a[i+3].z` and `b[i],...,b[i+3]` are stride-one accesses, their alignment constraints are handled no differently than any stride-one access of native data types.

For simplicity, Figure 3 assumes that both vectors are aligned. But consider in this paragraph that one of them is not, *e.g.*, say `a[0].y` is 16 byte aligned instead of `a[0].x`. Then, this phase would logically skew the computations associated with vector `a` to the right by one value. As a result, the blocked loop would effectively compute and store the `a[i].y,...,a[i+4].x` instead of `a[i].x,...,a[i+3].z`.

Length devirtualization. The `a[i].x,...,a[i+3].z` vector is broken down to 3 physical vectors as shown in Figure 3d. At this stage, all vectors have become physical vectors, namely they are aligned and operates on 16-byte data.

In this example, basic-block level and loop level simdization each handles some aspects of simdizing the loop:

```
unsigned int a[N];
unsigned short b[N];
for (int i = 0; i<N; i++) {
  a[i] = ...;
  b[i] = ...;
}
```

Figure 4: Mixed data length across statements.

```
uint a[N];
ushort b[N];
for (int i = 0; i<N; i++) {
  a[i] += (uint) b[i+1];
}
```

Figure 5: Mixed data lengths within a statement.

- Basic-block level simdization combines the otherwise non stride-one accesses, `a[i].x`, `a[i].y`, and `a[i].z` into stride-one accesses for subsequent loop level.
- Loop-level simdization aggregates the unaligned vector of 12 bytes, `a[i].x,y,z`, produced by basic-block level into an aligned vector of multiple of physical vector lengths.

Unlike conventional approaches, simdization of this loop is achieved transparently within the proposed framework without expensive loop restructuring such as loop rerolling and collapsing, or loop unrolling.

4. SIMDIZATION WITH MIXED DATA LENGTHS

In this section, we describe techniques to simdize loops with mixed data lengths.

4.1 Mixed Data Lengths

There are two types of loops that mix data of different lengths. The first type contains statements that each operates on data of uniform length, but different statements operate on data of different lengths. Figure 4 illustrates one such loop with type-homogeneous statements. The second type contains statements that operate on distinct data lengths within a single statement. Such statements often involve conversion operations between data of different lengths, referred to as *length conversion* in this paper. Figure 5 illustrates one loop with type-heterogeneous statements.

4.2 Integrated Loop Simdizer

In general, loops involving statements with length conversion are more constraining than loops with type-homogeneous statements. This is due to the fixed-length nature of SIMD vectors. For example, a simdized length conversion from 1-byte type length to 2-byte type length consumes 1 vector and produces 2 vectors. The property of consuming one vector but producing multiple vectors (or a fraction of a vector, the other way around) is unnatural to most compiler intermediate representations. We thus focus on this more challenging second case for the remainder of this section, using the loop in Figure 5 as a working example.

Loop level aggregation. This phase does not require uniform data type length involved in a computation. Thus,

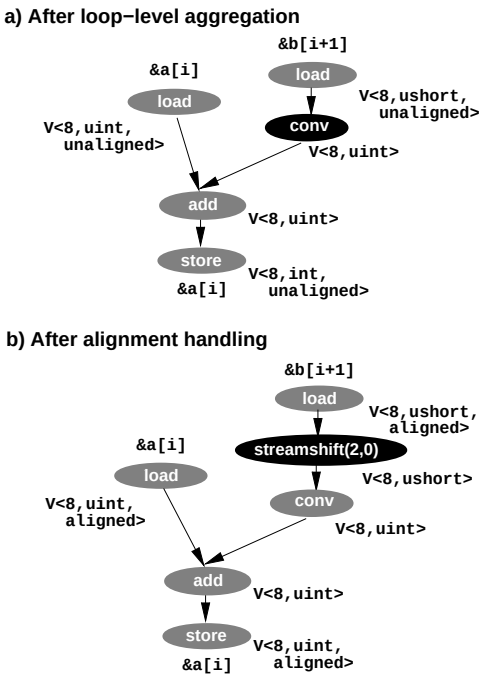

Figure 6: Length conversion after loop-level simdization.

a data conversion from type t_1 to type t_2, after loop-level aggregation, becomes a conversion between vector $V(B, t_1)$ and vector $V(B, t_2)$, where B is the blocking factor.

Consider the loop in Figure 5 which operates on data of short and int. Using formula (1), the blocking factor is 8, packing 8 `b[i]` shorts into a 16-byte vector and 8 `a[i]` integers into a 32-byte vector.

In Figure 6a, we give the expression tree after this phase. In the tree representation, each node is labeled with the tuple $V(n, t, aligned/unaligned)$ to represent, respectively, the number of elements, the type, and the alignment properties of each vector. Since the alignment property only affects vector load-store semantics, it is specified only for memory nodes.

Alignment devirtualization. This phase requires array accesses in the loop be stride-one across iterations in order to form a *stream* of contiguous memory accesses. Since the handling of length conversion during any of the three aggregation phases before does not change the stride-oneness of any accesses, stride-one accesses that are input to the previous phases still remain stride-one in the output.

Since length conversion is not supported in our original algorithm [6], we proposed an extension to incorporate length conversion into the alignment handling framework [7]. The new algorithm is mainly concerned with code generation issues with stream shift operations when length conversion is involved.

For our example, assuming the base addresses of `a` and `b` are aligned, the streams represented by `a[i]` and `b[i+1]` have an offset of 0 and 2, respectively, thus, are relatively misaligned. Figure 6b gives the expression tree after alignment handling. Node `streamshift(2,0)` represents a pseudo operation that "shifts" the entire input stream of `b[i+1]` with an offset of 2 to 0 to match the offset of `a[i]`

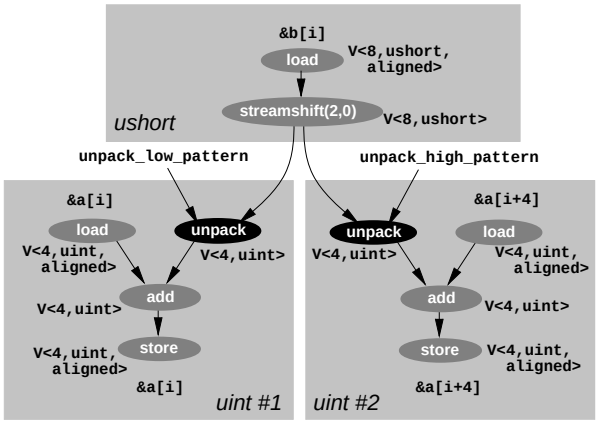

Figure 7: Length conversion after devirtualization.

stream.

Length devirtualization. For most vector operations, devirtualization simply involves breaking a vector into multiple physical vectors. Length conversion, however, requires special handling because a devirtualized length conversion may consume one physical vector and produces multiple physical vectors, or vice versa. We introduce two generic operations to express such data reorganization.

- `pack`$(v_1, v_2, v_{pattern})$. This operation produces a physical vector by selecting bytes from physical vector v_1 and v_2 according to the byte pattern specified by physical vector $v_{pattern}$.
- `unpack`$(v, v_{pattern})$. This operation produces a physical vector by selecting bytes from physical vector v according to the byte pattern specified by physical vector $v_{pattern}$.

For our example, the conversion operation unpacks a vector of short into two vectors of int. Figure 7 gives the expression trees after devirtualization. Note that, this simdized codes consist of three distinct parts. The top part represents the computations done in short, *i.e.*, mainly loading from memory. The bottom two parts represents the computations done in int, with the left and right bottom parts each consuming half of the data produced by the top part.

SIMD code generation. Generic `pack` and `unpack` operations are mapped to instructions of the target platform. For example, using VMX intrinsic, an `unpack` operation with some particular byte patterns can be mapped to `vec_unpackl` and `vec_unpackh`. Others can be mapped to `vec_perm` where the permutation pattern can be precomputed based on the unpacking factor.

We would like to point out that incorporating conversion handling into the simdization framework is made easier due to our ability to preserve stride-one accesses during early phases of simdization. As shown in Figure 7, when devirtualizing the `short` to `int` conversion, the stride-one access `a[i]` in the original loop becomes two *non* stride-one vector accesses, `a[i:i+3]` and `a[i+4:i+7]`. Since stride-one accesses are critical to both aggregation phases and alignment handling, by maintaining stride-oneness until devirtualization, these early simdization phases can process length conversion no differently from processing other operations.

```
for (j = 40; j <= 120; j++) {
    sum = 0;
    for (i=0; i<40; i++)
       sum += wt[i] * dp[i-j];
    L_result[j] = sum;
 }
```

Figure 8: A simplified FIR computation from `GSM`.

5. EXPANDING SCOPE OF INNERMOST-LOOP SIMDIZATION

In this section, we address the simdization of short loops that are particularly common in multimedia codes.

5.1 Constraints of loop-level simdization

One constraint of loop-level simdization is that it typically applies to the innermost loop only. In multimedia codes, however, there are many innermost loops with relatively short and often constant trip counts.

On the one hand, simdizing such short loops may sometimes degrade the performance due to the overhead associated with loop-level simdization. To give a few examples,

- Blocking a loop may produce residual loops that have to be executed sequentially. Blocking may also destroy perfect loop nest, thus affects subsequence loop transformations.
- When simdizing a reduction, the final sum reduction in the epilogue needs to be done in sequential mode.
- Alignment handling incurs overheads, such as setting up permutation masks and executing prologue and epilogue (loops) to handle partial stores, sometimes in sequential mode.

On the other hand, by restricting simdization to a single loop level, *i.e.*, the innermost one, one misses the opportunity to exploit SIMD parallelism across loop nests. Figure 8 gives an example of such a loop extracted and simplified from the `GSM` application from Berkeley multimedia Workload benchmarks[8].

5.2 Short loop aggregation

To expand the scope of loop-level simdization, our approach is to eliminate innermost loops with short trip count. The basic idea is to collapse simdizable short loops into vector operations, so that the next loop in the original loop nest becomes the new innermost loop. In doing so, we effectively expands the scope of subsequent simdization transformations. We refer to this technique as *short loop aggregation*.

The rest of the section illustrates our approach using the loop in Figure 8 as an example. First, we introduce several pseudo operations that are needed for this example.

- `reduct(`op`,`$V(n,t)$`)` performs a reduction (of operation op) over all n elements of vector V and generates a single reduction value.
- `parallel_reduct`$(op, x, V(n, V(x, t)))$ performs x reductions (of operation op) in parallel, where each reduction operates over n distinct values. The data in the vector is interleaved, *i.e.*, the first x values are the first elements in each of the x reductions, the next x values are the second elements in each of the reductions, and so on. The parallel reduction generates a vector of x reduction value.

- `elem_splat`$(V(n,t),x)$ generates a new $V(n,V(x,t))$ vector where each of the values in the original vector appears x times, consecutively.
- `elem_slide`$(V(n,t),w)$ generates a new $V(n,V(|w|,t))$ vector. In the new vector, the first $|w|$ values corresponds to the $0,\ldots,w-1$'th values in the original vector, and the i'th next $|w|$ values corresponds to the $i,\ldots,i+w-1$'th values in the original vector. This corresponds to applying a sliding window of w elements along the original vector. Note that w could be a negative (underflow/overflow values must also be carefully considered).

Short loop aggregation. The innermost i-loop in Figure 8 is simdizable with a reduction and two stride-one accesses, namely `wt[i]` and `dp[i-j]`. Since the loop has a trip count of 40, the loop can be simdized into a single statement operating on a vector of 40 elements. The loop nest after simdization is shown below.

```
for (j = 40; j <= 120; j++) {
  sum = 0;
  sum = sum + reduct(+,wt[0:39]*dp[-j:39-j]);
  L_result[j] = sum;
}
```

Loop-level aggregation. This phase simdizes the now innermost j-loop. It recognizes `sum` as a private variable, `L_result[j]` as a stride-one accesses, and `reduct` as a reduction over a vector. Assuming all data are `int`, the blocking factor can be computed as,

$$B = 16/\mathrm{GCD}(16, \texttt{sizeof(int)}) = 16/4 = 4.$$

Simdization of `sum` and `L_result[j]` is straightforward. But the handling of `reduct` is more complicated because data involved in the reduction are not stride-one across j-loop. Specifically, `wt[0:39]` is now loop-invariant (*i.e.*, stride zero) and `dp[-j:39-j]` has a stride of $-1/40$. The goal is to simdize `reduct` into a 4-way `parallel_reduct`.

Since `wt[0:39]` is of stride zero, to make the reduction a 4-way parallel reduction, each of its elements needs to be replicated 4 times, *i.e.*, `elem_splat(wt[0:39],4)`. For `dp[-j:39-j]`, the four vectors to be reduced in parallel are `dp[-j:39-j]`, ..., `dp[-j-3:39-j-3]`. Note that, the corresponding elements of the 4 data streams form a packed vector among themselves[2]. To produce a 4-way parallel reduction, each element of the original vector needs to be expanded by applying a sliding window of -4 elements, *i.e.*, `elem_slide(dp[-j:39-j],-4)`.

The code after loop-level simdization is shown below.

```
for (j = 40; i <= 120; i+=4) {
  vsum = (0,0,0,0);
  vsum = vsum + parallel_reduct(+, 4,elem_splat(
         wt[0:39])*elem_slide(dp[-j:39-j],-4));
  L_result[j:j+3] = vsum;
}
```

Devirtualization. Pseudo operation `parallel_reduct`, `elem_splat`, and `elem_slide` are mapped to operations on physical vectors. Since direct handling of non stride-one accesses is not the focus of the paper, the mapped codes are omitted.

In this example, by aggregating i-loop into a single vector statement, we are able to simdize the i-j loop nest together. Most important of all, the expanded simdization scope allows more efficient ways of extracting SIMD parallelism. For example, we are able to exploit the reuse of `wt[i]` and `dp[i-j]` across j-loop, manifested as sub-one strides. It also enables the transformation of a horizontal reduction to a more efficient vertical parallel reduction.

On the other hand, even if we fail to simdize j-loop after short loop aggregation, we can still safely simdize the `reduct`, which is the original innermost loop computation.

6. DEVIRTUALIZATION AND MIXED-MODE SIMDIZATION

We can further extend the concept of virtualization to vector operations. In essence, during early phases of simdization, an operation can be aggregated into a vector operation even though the operation is not supported by the hardware. At the devirtualization phase, the final decision is made on whether to map a vector operation to a single SIMD instruction, a sequence of SIMD instructions, or a sequence of scalar operations, based on a cost model. Since this scheme enables us to generate a mixture of SIMD and scalar codes within a loop, we refer to this code generation scheme as mixed-mode simdization.

We first discover the need of mixed-mode simdization when we realize the "heterogeneous" nature of computation involved in a loop. For example, a loop may contain computation that can be efficiently executed on the scalar unit only, on the SIMD unit only, or on both. Using VMX as an example, double-float computation must be executed on scalar units, saturate arithmetic is better performed on a SIMD unit, while most fixed point operations can be computed on both units. Furthermore, since SIMD units are in general more constrained than the scalar ones, sometimes it may be more efficient to perform computation in scalar mode if accesses are misaligned or not stride-one.

One approach to simdize heterogeneous loops is to distribute the loop so that one contains all operations to be executed on the vector unit, and the other with operations to be executed on scalar units. The major drawback of loop distribution is that it creates more loops with shorter loop bodies. This works against the exploitation of instruction-level parallelism (ILP), as it decreases the size of basic blocks. In fact, overly aggressive loop distribution is one of the major causes of simdization related performance degradation.

With mixed-mode simdization, we not only avoid hurting ILP as loop distribution does, but may further enhance ILP as the scheduler can now utilize both SIMD and scalar units and registers at the same time. Sometime, it is not straightforward to decide on which unit the processing is more beneficial. Hence, one must take into account both hardware constraints and resource constraints of current program execution when deciding how to distribute computation across SIMD and scalar units. Furthermore, one may exploit both VMX and scalar units for the same operation at the same time. For example, one can issue 1 vector add and 1 scalar add at the same time as if executing a SIMD add on a 5-element-wide SIMD unit.

[2]This can be automatically deduced based on the stride of the original vector. Because each of the 4 streams of data to be reduced in parallel is offseted by $-1/40$ of the vector type $V(40,\texttt{int})$, *i.e.*, by the length of an `int`, thus is of stride -1 among corresponding elements across different data streams.

Furthermore, since there are different ways to distribute computation between SIMD and scalar units depending on the context of the execution, devirtualization provides us the flexibility to revert some of the less optimal local decisions made earlier when more information is available. For example, during basic-block level simdization, one may pack non stride-one accesses into vectors, hoping that these vectors may become stride-one after loop-level aggregation. If that is not the case and the packing overhead is too significant, however, we can still revert the overly optimistic packing decision by devirtualizing the vector operations back to scalar operations.

7. EXPERIMENTAL RESULTS

The proposed simdization framework is implemented in IBM's XL product compiler, which supports multiple programming languages and generates highly optimized codes for multiple target machines. Prior to simdization, a wide range of high-level optimizations are performed, such as loop distribution and loop versioning. Other high-level optimizations are applied after simdization as well as traditional backend optimizations.

Our simdization infrastructure is rapidly maturing, currently simdizing innermost loops with stride-one memory accesses as well as induction, reduction, and private variables. It performs interprocedural alignment analysis. In terms of alignment handling, it handles loops of multiple statements with arbitrary misalignments. It supports loops with mixed data types. It supports basic-block and loop level simdization, as well as a preliminary version of short loop aggregation.

Measurements are performed on a IBM PowerPC PPC970 processor found in Apple PowerMac G5 with full SIMD Vector Multimedia eXtension (VMX) support and running under MAC OSX. In addition to the 5 scalar units (2 fix point, 2 floating point, and 1 branch), the PPC970 provides 2 SIMD VMX units (1 arithmetic and 1 permute) plus 32 additional 16-byte wide vector registers. The SIMD units support vector of `char`, `short`, `int`, and `float`. Two memory units are shared between the scalar and vector units. In vector mode, the memory units support only 16-byte aligned memory accesses.

7.1 Kernel Results

We report in Figure 9 the simdization speedup factors achieved on six scientific and multimedia kernels. Performance is reported as a speedup factor achieved by the automatically simdized loop over the sequential version of the same loop compiled at the same optimization level. Time is measured with hardware counters and includes any branching overhead, address computation overhead, and any extra code introduced to setup the simdized computations.

The kernels are extracted from real applications that include numerical applications (`blas` routines), video applications (`alpha-blending`, `subdivide`), and communication applications (`TCP/IP` and `fir`). We give a more detailed description of these kernels and their simdization below.

blas.sdotprod is a numerical dot product routine computing `A*B` where `A` and `B`are single precision vectors. On the PPC970, we would expect up to a factor of 2 speedup improvement (a 4-way SIMD floating point unit vs. 2 scalar floating point units). We achieve a speedup of 1.74, less than peek due to the more limited addressing modes of the VMX units with respect to the scalar units.

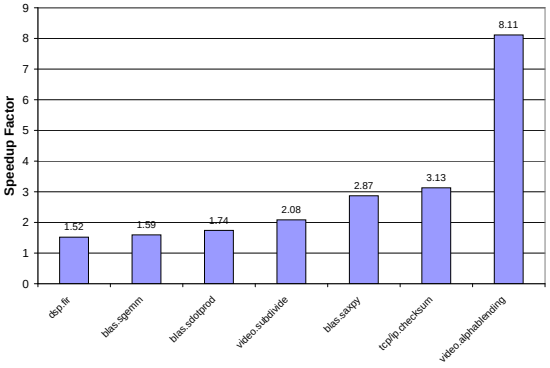

Figure 9: Achieved simdization speedups for various kernels.

Kernel	Alignment	Features	Types
blas.sdotprod	none	reduction	float
blas.sgemm	none		float
blas.saxpy	none		float
dsp.fir	runtime	reduction	short→int
tpc/ip.checksum	runtime	reduction	short→int
video.subdivide	none	basic block SIMD	float
video.alphablend	none		char→short short→char

Figure 10: Characteristics of kernels.

blas.sgemm is a numerical matrix routine computing `alpha*A*B+beta*C` where `A`, `B`, and `C` are single precision matrices. The ratio of scalar vs SIMD floating point units indicates a peek speedup of 2. The somewhat smaller speedup achieved (1.59) is primarily due to the presence of control flow codes dispatching to different special cases.

blas.saxpy is a numerical vector routine computing `alpha*X+Y` where `X` and `Y` are single precision vectors. While up to a speedup factor of 2 can be achieved on the computation, memory intensive kernels such as this can achieve up to a factor of 4 on loads and stores. The achieved speedup, 2.87, is in between the two bounds.

dsp.fir is a filtering routines often found in DSP and communication applications (*e.g.*, GSM), similar to the one shown in Figure 1. This particular instance has 40 taps and process an input of 120 short values. The reduction is performed as integer (requiring length conversion) in parallel, and its output is scaled down back to a short. Only the innermost loop is simdized presently. While a speedup factor of up to 2 could be achieved for the integer reduction (one 4-way SIMD unit vs. 2 scalar units), we get a factor of 1.52. This lower number is due to the mandatory realignment that occurs as the inputs of each consecutive fir value cycle through all possible data alignments.

tcp/ip.checksum is a communication routine that computes the checksum of incoming network packets to verify the integrity of the input data. This particular instance computes an integer checksum of a 4K char buffer by accumulating short values. Like `dsp.fir`, this kernel stresses data conversion and reduction. In addition, this kernel uses

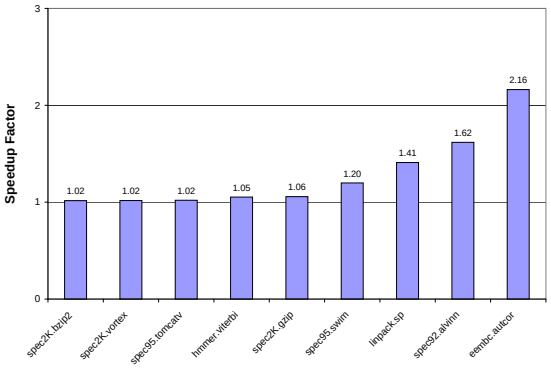

Figure 11: Achieved simdization speedups for various benchmarks.

extensive pointer arithmetic (that needs to be reverted by the compiler) and passes the buffer of 16-bit shorts as an array of 8-bit char. Thus, we may not assume that the 16-bit shorts fall at natural 16-bit boundaries, requiring extra alignment operations. It achieve a speedup factor of 3.13, higher than `dsp.fir` mainly because the reduction's overhead are amortized over a larger data set than `dsp.fir`.

video.subdivide is a graphics routine that subdivides triangle representations. It is a memory and computation intensive kernel that requires 64 bytes of data to produce a new 16 byte triangle with 16 single precision floating point multiply and add. The computations are expressed as isomorphic statements over 4 elements of a struct, thus requiring basic block aggregation to simdize its innermost loop. It achieves a 2.08 speedup factor, which will be further improved once we succeed in removing a redundant struct copy.

video.alphablending is a graphics routine that blends pictures with variable amount of transparency. Unlike the previous kernels that processed 16 and 32 bit quantities, this kernel processes 8-bit char quantities, where a subset of the computations is performed over 16-bit shorts. It thus requires conversion from `char` to `short`, then followed by conversion from `short` back to `chars`. It achieves a 8.11 speedup factor, which is between the maximum factor of 8 and 16 associated with, respectively, short computation and char memory accesses.

7.2 Application Results

We report in Figure 11 the speedup factors (ranging from 1.02 to 2.16) achieved by simdizing 9 benchmark applications. In general, the speedup factors are more modest, as only part of applications are suitable for simdization. Also, simdization and the generation of VMX codes are stressing new parts of the compiler's backend, requiring further fine tuning in order to match the expected performance improvements.

At the present time, the following benchmarks achieve minor speedup factors. The bzip and vortex spec2000 benchmarks achieve minor speedup because the simdized loops do not significantly contribute to the overall computation. The tomcatv benchmark was transformed to operate over single precision and achieve only moderate performance improvement. Many of the simdized loops are fairly computation intensive, with several of the inputs data reused in multiple sub-computation. We believe that at the present time, the backend cannot reused the simdized input as well, thus resulting in spurious loads and data reorganizations. Finally, only a few loops of viterbi are simdized.

The next benchmarks indicate more promising speedup factors. In `Gzip`, we successfully simdize a loop performing saturating arithmetic. In swim, we simdize many of the main loops (except the ones with divide), after converting it to use single precision floats. We get a speedup factor of 1.41 in linpack in single precision mode. In alvinn, we also simdize many of the important loops. In `Autocor`, a 2.16 speedup is achieved by successfully simdizing the loop that involves runtime alignment, length conversion, and reduction.

8. RELATED WORK

There has been a recent upsurge of compiler research on automatic vectorization for SIMD units [4, 5, 9, 10, 11, 12, 13]. Two principal simdization approaches have been used, the traditional loop-based vectorization [9, 12, 14, 15] pioneered for vector supercomputers [16, 17] and the basic-block approach [5, 18, 19] first proposed by [5]. To exploit SIMD parallelism at both levels, most prior work choose one approach as their simdization method, then convert one form of SIMD parallelism to another. There are three commonly used techniques.

- *Loop rerolling* converts basic-block level parallelism to loop-level by rerolling isomorphic statements into loops[20]. The drawback of loop rerolling is the introduction of a new innermost loop into the loop nest. Since loop-level simdization happens at the innermost level, rerolling does not expand the scope of simdization, but rather exploits basic-block level parallelism in loop forms. Furthermore, rerolling requires the presence of a loop counter, which may not always possible.
- *Loop collapsing* rerolls isomorphic statements into a loop and then collapses it to the innermost surrounding loop[9]. Loop collapsing is able to exploit parallelism at both levels. However, it is also more constraining as it requires the original innermost loop to be perfectly nested to its surrounding loop and all its statements be rerolled. For example, it can not collapse the loop shown in Figure 3.
- *Loop unroll-and-pack* converts loop-level parallelism to basic-block level parallelism by first unrolling the loop, then packing computation within a basic block into SIMD instructions [5, 18]. Since simdization is limited to a basic block, it can be very inefficient to handle misalignment and reduction. Furthermore, packing algorithms (either greedy- or search-based) tend to be more expensive and randomized, and sometime less optimal than loop-level vectorization which are often based on dependence vectors.

Compared to the above approaches, our scheme is both more flexible and more powerful, combining the merits of both approaches. Our scheme is also less expensive because it requires no change to loop nest structures. Most important of all, it integrates nicely with other aspects of simdization, such as alignment handling.

For loops with mixed data lengths, many compilers resort to loop distribution to construct loops with a uniform data length, such as the Intel compiler [4]. For length conversion, many compilers either do not simdize the loop or

only simdize length conversions that are directly supported by the hardware, such as a factor of 2 packing/unpacking [4]. The VAST compiler [20] appears to handle both types of mixed data length loops. However, there is no public information available of their technique. In our framework, we are able to support either types of loops with mixed data lengths, and length conversion with arbitrary conversion factors. We also support length conversion with misalignment.

9. CONCLUSION AND FUTURE WORK

There has been increasing complexity in automatic simdization for multimedia extensions. This is mainly due to the need to address multiple orthogonal simdization issues both independently and collaboratively.

In this paper, we propose a simdization framework that seamlessly integrates different aspects of simdization, *e.g.*, alignment handling, length conversion, and multiple levels of SIMD parallelism extraction. Our strategy is to allow earlier simdization phases to operate on virtual vectors, which are less constrained than the physical vectors supported by the hardware. Individual architectural constraints are then handled in succession in later simdization phases, gradually converting virtual vectors to physical vectors.

Measurements on a PPC970 with a VMX SIMD unit indicate speedup factors of up to 8.11 for numerical/video/communication kernels and speedup factors of up to 2.16 for benchmarks with automatic simdization.

Many further issues need to be investigated before we can enjoy the performance benefit of simdization for a wide range of applications. The more important features among them are more advanced handling of non stride-one accesses and the ability to decide when simdization is profitable. Equally important is a better understanding of the interaction between simdization and other optimizations in a compiler framework.

10. REFERENCES

[1] IBM Corporation. PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual, July 2004.

[2] *IA32 Intel Architecture Software Developer's Manual with Preliminary Intel Pentium 4 Processor Information Volume 1: Basic Architecture*. Intel Corporation.

[3] B.Flachs *et al.* A Streaming Processing Unit for a CELL Processor. In *IEEE International Solid-State Circuits Conference*, February 2005.

[4] Aart J.C. Bik. *The Software Vectorization Handbook*. Intel Press, 2004.

[5] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[6] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 2004.

[7] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of International Symposium on Code Generation and Optimization*, March 2005.

[8] Nathan T. Slingerland and Alan Jay Smith. Design and Characterization of the Berkeley Multimedia Workload. *Multimedia Systems*, 8(4):315–327, July 2002.

[9] Aart Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *International Journal of Parallel Programming*, (2):65–98, April 2002.

[10] Gerald Cheong and Monica S. Lam. An Optimizer for Multimedia Instruction Sets. In *Second SUIF Compiler Workshop*, August 1997.

[11] Andreas Krall and Sylvain Lelait. Compilation Techniques for Multimedia Processors. *International Journal of Parallel Programming*, (4):347–361, August 2000.

[12] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a SIMdD DSP Architecture. In *Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 2–11, October 2003.

[13] Free Software Foundation. http://gcc.gnu.org/projects/tree-ssa.

[14] N. Sreraman and R. Govindarajan. A Vectorizing Compiler for Multimedia Extensions. *International Journal of Parallel Programming*, 28(4):363–400, August 2000.

[15] Corinna G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of International Symposium on Microarchitecture*, pages 25–36, 1998.

[16] John Randal Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, (4):491–542, October 1987.

[17] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[18] Franz Franchetti, Stefan Kral, Huergen Lorenz, and Christoph Ueberhuber. Efficient utilization of SIMD Extensions. In *IEEE Proceedings Special Issue on Program Generation, Optimization, and Platform Adaptation*, 2005.

[19] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[20] Crescent Bay Software. VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit. `http://www.psrv.com/vast_altivec.html`, 2004.