Efficient SIMD Code Generation for Runtime Alignment and Length Conversion

Peng Wu Alexandre E. Eichenberger Amy Wang

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

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

Abstract

When generating codes for today's multimedia extensions, one of the major challenges is to deal with memory alignment issues. While hand programming still yields best performing SIMD codes, it is both time consuming and error prone. Compiler technology has greatly improved, including techniques that simdize loops with misaligned accesses by automatically rearranging misaligned memory streams in registers. Current techniques are applicable to runtime alignments, but they aggressively reduce the alignment overhead only when all alignments are known at compile time.

This paper presents two major enhancements to the state of the art, improving both performance and coverage. First, we propose a novel technique to simdize loops with runtime alignment nearly as efficiently as those with compile-time misalignment. Runtime alignment is pervasive in real applications because it is either part of the algorithms, or it is an artifact of the compiler's inability to extract accurate alignment information from complex applications. Second, we incorporate length conversion operations, e.g.*, conversions between data of different sizes, into the alignment handling framework. Length conversions are pervasive in multimedia applications where mixed integer types are often used. Supporting length conversion can greatly improve the coverage of simdizable loops. Experimental results indicate that our runtime alignment technique achieves a 19% to 32% speedup increase over prior art for a benchmark stressing the impact of misaligned data. We also demonstrate speedup factors of up to 8.11 for real benchmarks over sequential execution.*

1 Introduction

Multimedia extensions have become a popular addition to most general-purpose micro-processors as they provide significantly increased processing power at a moderate hardware cost. Existing multimedia extensions are characterized as **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) units that operate on packed, fixed-length vectors, such as MMX/SSE for Intel and VMX/AltiVec for IBM/Apple/Motorola. Similar SIMD units can also be found in graphics engines, game consoles, and DSP processors.

While SIMD units are becoming ubiquitous, their impact on mainstream program performance is still under their full potential. This is mainly because of the difficulties to produce SIMD codes either manually by hand or automatically by compilers (referred to as **simdization** thereafter). Some of these difficulties are unique to simdization because they are the direct result of the more restrictive hardware constraints imposed by today's SIMD architectures [12].

The alignment constraint of SIMD memory units is such a hardware feature that can significantly impact the effectiveness of simdization. For example, the memory operations in AltiVec [5] and others can only access 16-byte contiguous memory from 16-byte aligned addresses. To demonstrate the implication of alignment constraints to simdization, let us consider the simplistic example in Figure 1, where the bases of arrays `a`, `b`, and `c` are aligned.

```
for (i=0; i<100; i++) {
  a[i+2] = b[i+1] + c[i+3];
}
```

Figure 1. A loop with misaligned accesses.

As illustrated in Figure 2, due to the alignment constraints of SIMD memory instructions, data involved in the same computation, *i.e.*, `a[i+2]`, `b[i+1]`, `c[i+3]`, are *relatively misaligned* after being loaded to registers. To produce correct results, these data must be reorganized to reside in the same slot of their corresponding registers prior to performing any arithmetic computation.

Our recently proposed simdization technique [6] and the VAST compiler [13] are able to simdize the loop in Figure 1. Both analyze the alignment of each memory reference and *automatically reorganize* data in registers to be aligned to each other. Figure 2 illustrates a successful simdization of the code in Figure 1 using our algorithm. The realigning of data in registers is achieved by shifting right by one value the stream of consecutive data originated from `b[i+1]` for

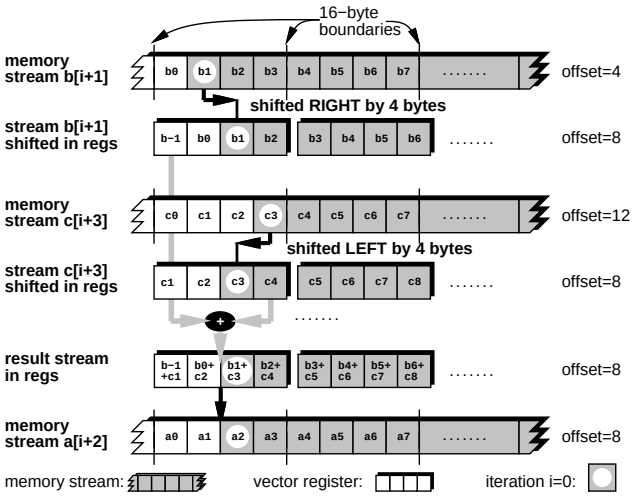

Figure 2. Simdization of `a[i+2]=b[i+1]+ c[i+3]` with minimum data reorganizations.

`i=0` to `99`, and by shifting left by one value the stream of data accessed by `c[i+3]` for `i=0` to `99`. The vector add is then applied to the shifted streams and produces the expected results, `b[1]+c[3]`, `b[2]+c[4]`,.... To understand the applicability of this scheme, it is critical to realize that "shifting left" and "shifting right" in this context are data reorganization operations that operate on a stream of consecutive registers, not the traditional arithmetic/logical shift operation.

To highlight the challenges faced when simdizing real applications, let us consider the two code fragments in Figure 3 and Figure 4, which are similar in nature to code found in `swim`, a SPEC95 benchmark, and `autcor`, an EEMBC/Telecom benchmark, respectively.

- The first challenge deals with the unavailability of memory alignment at compile-time (referred to as **runtime alignment**). Runtime alignment can happen in different contexts. For example, it can be inherent to the program as is the case in Figure 3, where all arrays are of dimensions 513 by 513, making alignment of every array access in Figure 3 runtime. More likely, runtime alignment can occur in complex pointer codes where the compiler is unable to extract accurate alignment information interprocedurally.
- The second challenge deals with data size conversions (referred to as **length conversion**). Such data conversions are very common in multimedia workload because multimedia data (*e.g.*, pixels or colors) are often stored in compact forms such as shorts or chars but are operated on as integers to reduce rounding errors. For example, in Figure 4, the input array `b` is of 16-bit short type and has to be extended before being accumulated into a 32-bit integer. Note that the alignment of `b[i+j]` changes for each `j` outer-loop iteration, making the alignment runtime.

```
DO 200 J=1,N
 DO 200 I=1,M
   UNEW(I+1,J)=UOLD(I+1,J)+T8*(Z(I+1,J+1)+
 1  Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)+
 2  CV(I,J)+CV(I+1,J))-TX*(H(I+1,J)-H(I,J))
   VNEW(I,J+1)=VOLD(I,J+1)-T8*(Z(I+1,J+1)+
 1  Z(I,J+1))*(CU(I+1,J+1)+CU(I,J+1)+
 2  CU(I,J)+CU(I+1,J))-TY*(H(I,J+1)-H(I,J))
   PNEW(I,J)=POLD(I,J)-TX*(CU(I+1,J)
 1  -CU(I,J))-TY*(CV(I,J+1)-CV(I,J))
200 CONTINUE
```

Figure 3. Runtime alignment in SPEC95 SWIM.

```
for (j=0; j<N; j++) {
  int a = 0;
  for (i=0; i<M-j; i++)
    a += ((int)b[i]*(int)b[i+j])>>n;
  ...
}
```

Figure 4. Length conversion in EEMBC autcor.

In this paper, we address two major limitations of our prior alignment handling technique [6] regarding these two challenges.

- The first limitation is that prior work [6] cannot minimize the number of data reorganizations when dealing with runtime alignment. This is because when generating codes for stream shifts, the compiler must know the direction of a shift, *i.e.*, whether it is a stream shift to the left or right. To accommodate this code generation constraint, the compiler has to resort to a less optimized data reorganization scheme in presence of runtime alignment.
- The second limitation is that prior work [6] does not handle misalignment in the presence of length conversions. A unique property of length conversions is its multiplying effect to data length as if computations are performed on vectors with variable lengths. This violates a fundamental assumption of the previous alignment handling framework, that is, all operations are performed on a *uniform* vector length.

We propose here a novel technique that addresses both the runtime alignment and the length conversion issues outlined above. The key insight to our solution is that the original memory and computations streams present in a loop can be transformed into equivalent streams that have nicer properties. This in turn enable us to shift them to arbitrary alignments even in the presence of runtime alignment and length conversion. The main contribution of this paper is to detail this transformation step and demonstrate that the same code generation techniques as in prior work [6] can

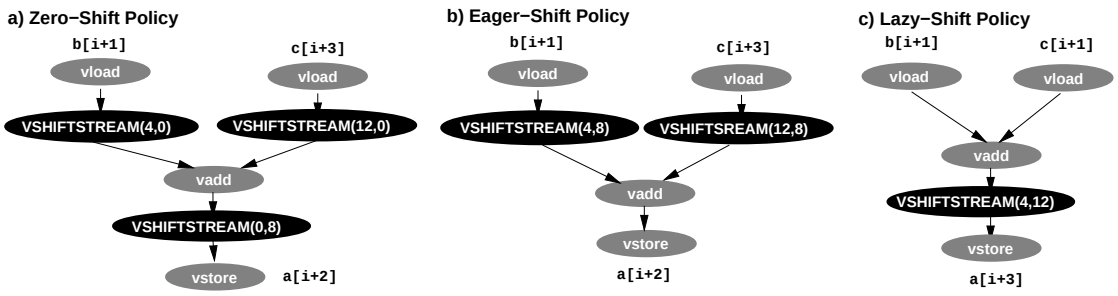

Figure 5. Simdization using different stream shift policies.

then be safely applied to the transformed streams. To the best of our knowledge, this is the first framework that is capable of generating efficient codes for arbitrary stream shifts with runtime alignment and length conversion.

Performance evaluation indicates that our technique significantly reduces the overhead of runtime alignment handling during simdization. Comparing performance speedups between simdized and scalar codes generated by our production compiler, we achieve the following speedups of simdization speedup factors over a wide range of loop benchmarks on a Apple Mac G5 with VMX unit. With 4 integers packed in a vector register and with two loads and one store per loop, all of which have runtime alignment, we improve simdization speedup factors from 2.7 (prior art) to 3.2 (our technique), achieving a 19% improvement. We achieve similar improvements over other data types, 34% for 8 shorts packed into a vector and 21% for 16 chars packed into a vector. We also demonstrate significant speedup factors for real benchmarks over sequential execution, *e.g.*, achieving factors of 2.16 and 8.14 for the entire `autocor` and `alphablend` applications, respectively.

The remaining of this paper is organized as follows. Section 2 gives an overview of the alignment handling framework in [6]. We then describe our runtime alignment handling technique in Section 3. Section 4 further extends our framework to incorporate length conversion. Then Section 5 integrates everything into our original alignment handling framework. Experiment results are presented in Section 6. Section 7 discusses the related work and we conclude in Section 8.

2 Background

In this section, we give an overview of the alignment handling framework in previous work and highlight some of the key concepts that our technique is based upon.

2.1 Basics Definitions

The alignment handling framework in [6] is based on the concept of streams. A **stream** represents a sequence of contiguous memory locations that are accessed by a memory reference throughout the lifetime of a loop (as shown in Figure 2 as a sequence of grey boxes). By analogy, a stream is also a sequence of contiguous registers that are produced by an operation over the lifetime of a loop. Vector operations in a loop can be viewed as operations over streams. For example, a vector load consumes a stream of memory and produces a stream of registers. An important property of streams is its **stream offset**. It is defined as the byte offset of the first desired value in the first register of a stream. Note that the offset of a register stream produced by a vector load is the alignment of the first desired value of the input memory stream (namely the memory address of the first desired value modulo the vector length of the SIMD unit). Figure 2 shows the stream offsets on the right hand side of each stream.

Based on the concept of streams, this framework specifies the **alignment constraints of a valid simdization** as follows:

- When simdizing a store operation, the byte offset of the data in the vector register must match the memory alignment of the store address. In other words, the offset of the register stream being stored must match the alignment of the store memory stream.
- When simdizing a non-unary operation, data involved in the original computation must reside at the same byte offset in their respective vector registers. In other words, streams involved in the same SIMD operation must have matching offsets.

In the presence of misalignments, a valid simdization can only be achieved by judiciously inserting data reorganization operations to enforce the desired stream offsets. The **stream shift** operation `vshiftstream(`S, c`)` is introduced for this purpose. It shifts all values of a register stream S across consecutive registers of the stream to an offset of c. Figure 2 gives examples of shifting streams left and right.

2.2 Overview of Framework

The alignment handling framework is based on placing stream shifts to satisfy the alignment constraint of a valid simdization. It consists of the following two phases.

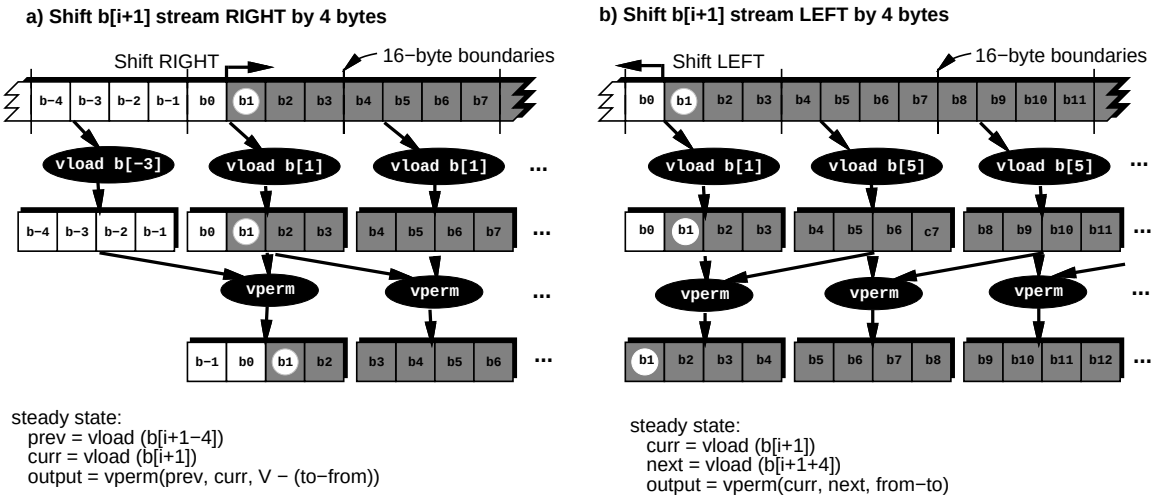

Figure 6. Code generation for shift stream left and right.

Phase 1: Data reorganization. This phase takes an expression tree as input and inserts shift operations to satisfy the alignment constraints of SIMD operations. The output is a tree augmented with stream shift operations, such as the ones shown in Figure 5. During this process, different shift placement policies can be applied to minimize the number of shifts generated. Here we only highlight three shift policies that are most relevant to this paper:

Zero-Shift Policy. This policy shifts each misaligned load stream to offset zero, and shifts the store stream from offset zero to the alignment of the store address. Figure 5a shows its resulting simdization for the loop in Figure 1. This is the least optimized policy and is the default policy all the time [13] or for runtime alignment [6].

Eager-Shift Policy. This policy shifts each load stream directly to the alignment of the store. Figure 5b illustrates its resulting simdization for the loop in Figure 1. Its corresponding execution trace is shown in Figure 2.

Lazy-Shift Policy. This policy pushes the shift towards the root of the expression tree as close as possible. Figure 5c illustrates its resulting simdization for loop `a[i+3]=b[i+1]+c[i+1]`.

In general, Eager-Shift inserts fewer stream shifts than Zero-Shift, *e.g.*, 2 versus 3 for the loop in Figure 1. Until now, however, Eager-Shift is only applicable to compile-time alignment [6] due to code generation issues alluded to in the introduction and expanded below. This makes the handling of runtime alignment in prior work less efficient.

Phase 2: Code generation. This phase takes the augmented tree as input and maps generic stream shift operations to native SIMD permutation instructions. In Figure 6, we only illustrate the most basic code generation scheme for stream shift[1]. For each stream shift in the tree, the algorithm generates a `vperm` instruction in the generated steady-state loop. Specifically, a `vperm`(v_1, v_2, ℓ) selects bytes $\ell, \ell+1, \ldots, \ell+V-1$ from a double-length vector constructed by concatenating v_1 and v_2, where V is the vector length. For example, on AltiVec, `vperm` is mapped to a `vec_perm` instruction.

Please note two important code generation features that are relevant to this work. First, different code sequences are generated depending on the direction of the shifts. Indeed, shifting streams to the right, like in Figure 6a, requires the code to combine the values from the *current* register with values from the *previous* register. Contrast this to the code sequence for a shifting a stream left, like in Figure 6b, where the code must combine the values from the *current* register with values from the *next* register.

Second, our algorithm exploits the fact that SIMD loads such as `vload b[i+1]` have their address truncated to the nearest V address when performing a SIMD load operation.

3 Efficient Runtime Alignment Handling

In this section, the examples have compile time alignments for illustration purpose. However, the proposed algorithm is suitable for runtime alignment because we demonstrate that our algorithm never uses specific alignment information at compile time.

3.1 Problem Illustration and Overall Approach

While Eager-shift works well for stream offsets known at compile time, it does not work for runtime alignment for

[1] Please refer to [6] for details on how to reuse loads feeding to consecutive `vperm`, handling of partial store, unknown loop bounds, and multiple statements with multiple misalignments.

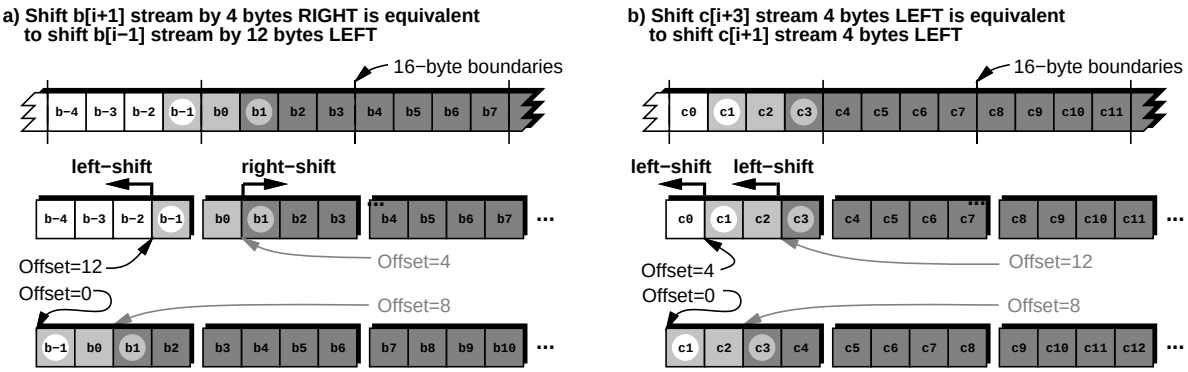

Figure 7. Converting arbitrary stream shift to an equivalent stream left-shift.

the following reason. As indicated in Section 2.2, the code sequence used for shifting streams left or right are different. The problem with runtime alignment is that the compiler does not generally know the direction of the stream shifts at compile time. Indeed, shifting a stream from arbitrary runtime offsets x to y corresponds to a right-shift when $x \leq y$, and a left-shift when $x \geq y$. Thus the compiler is restricted to apply the Zero-shift policy to runtime alignment since the direction of shifting from x to 0 or from 0 to y can always be determined at compile-time.

The key insight is to realize that this code generation problem occurs because we are focusing on the wrong element of the stream. We show in this section that by focusing on a different element of the stream (mechanically derived from the runtime alignment y), we can use a left stream shift code sequence regardless of the alignments x or y. Instead of focusing on the first element of the stream, we focus here on the element that is both at offset zero *after* shifting the stream and in the *same* register as the original first value.

Consider our initial example of `a[i+2]= b[i+1]+c[i+3]` to illustrates the first values and the new values of interest for streams `b` and `c`. Using the simdization based on the Eager-Shift policy shown in Figure 5b, this simdization invokes a right-shift for `vload b[i+1]` from offsets 4 to 8. As shown in Figure 7a, the initial value of the stream is `b[1]`. The new value of interest is `b[-1]`, namely the value at offset zero in the shifted register that also contains `b[1]`. Similarly, this simdization invokes a left-shift for `vload c[i+3]` from offsets 12 to 8. As shown in Figure 7b, the initial value of the stream is `c[3]` and the value at offset zero in the first shifted register is `c[1]`.

Let us now derive two new streams, which are constructed by prepending a few values to the original `b[i+1]` and `c[i+3]` streams so that the new streams start at, respectively, `b[-1]` and `c[1]`. These new streams are shown in Figures 7 with the prepended values in light grey and the original values in dark grey. Using the same definition of the stream offset as before, the offsets of the new memory streams are 12 and 4, respectively.

Consider now the result of shifting the newly prepended memory streams to offset zero. As shown in Figures 7a and 7b, the shifted new streams yield the same sequence of registers as that produced by shifting the original stream (highlighted with dark grey box with light grey circle), as the first values of the original streams, `b[1]` and `c[3]`, land at the desired offset 8 in the newly shifted stream. This holds because the initial values of the new streams were selected precisely as the ones that will land at offset zero in the shifted version of the original streams. Since shifting any stream to offset zero is a left stream shift, by definition, we have effectively transformed an arbitrary stream shift into a left-shift, as shown in Figures 7a and 7b.

3.2 Stream Prepend and Stream Skip

In this section, we formally introduce the **stream prepend** operator, which serves as a powerful tool to transforming stream shifts. We use $\texttt{Prepend}_W(S, x)$ to denote prepending x bytes to the beginning of a stream S of W-byte wide registers. Note that, the W associated with prepend is a free variable that can be set to any value. Its introduction is mostly for the handling of length conversion (in the next section). For this section, we always set the value of W to be V. However, when describing general properties of prepend, we will still use W instead of V.

Prepending a stream S by x bytes is only technically defined for memory streams, where one simply subtracts x bytes from the memory addresses. Prepending to non-memory streams, namely to register streams, is not directly feasible. However, we provide rules that propagate the prepend operation from an arbitrary register stream to the leaves of its expression tree until all of its memory streams are reached.

We describe below how to perform/propagate stream prepend for the three type of nodes (memory, register computation, and shift stream) present in an expression tree.

- $\text{VLOAD}(addr(i))$. This node represents a vector load from a stride-one access $addr(i)$ where $addr(i)$ is

truncated at V-byte boundary. Therefore,

$$\texttt{Prepend}_W(\text{VLOAD}(addr(i)), x) \Rightarrow \text{VLOAD}_W(addr(i) - x). \quad (D.1)$$

where VLOAD_W represents a vector load that truncates at W-byte boundary.

- $\text{VOP}(S_1 \ldots S_n)$. This node represents a generic operation that takes as input register streams $S_1 \ldots S_n$ and produces one output register stream. Thus,

$$\texttt{Prepend}_W(\text{VOP}(S_1 \ldots S_n), x) \Rightarrow \text{VOP}(\texttt{Prepend}_W(S_1, x) \ldots \texttt{Prepend}_W(S_n, x)). \quad (D.2)$$

- $\text{VSHIFTSTREAM}(S, to)$. This node shifts the register stream S to offset to, producing a register stream with a stream offset to. Thus,

$$\texttt{Prepend}_W(\text{VSHIFTSTREAM}(S, to), x) \Rightarrow \text{VSHIFTSTREAM}(\texttt{Prepend}_{W'}(S, x), (to - x) \bmod W). \quad (D.3)$$

According to the definition of prepend specified in Equations (D.1) to (D.3), one can prove the following properties of a prepended stream:

$$\mathit{Offset}(\texttt{Prepend}_W(S, x)) = (\mathit{Offset}(S) - x) \bmod W \quad (4)$$

$$\mathit{Length}(\texttt{Prepend}_W(S, x)) = \mathit{Length}(S) + x \quad (5)$$

Due to space constraints, the proof of Equations (4) and (5) are omitted.

We also introduce an inverse of the prepend operation, $\texttt{Skip}_W(S, x)$, which skips the first x bytes of stream S of W-byte wide registers. While we can derive similar properties for the skip operator as the ones derived for the prepend operator, we use the skip operator mainly for book-keeping purpose and thus will not need such relations.

3.3 Transforming Stream Shift to Left-Shift

As illustrated in Section 3.1, an arbitrary stream shift $\texttt{vshiftstream}(S, x)$ can be converted to a left stream shift to offset zero of a derived stream, one that starts exactly x bytes before the first value of S.

Theorem 1. An arbitrary register stream S can be shifted to an arbitrary target offset to by 1) prepending to bytes to a stream S of V-byte registers, 2) left-shifting the prepended stream to offset zero, and 3) skipping the first to bytes from the resulting stream. No code is required for Step 3 as it is only a book-keeping, nop operation.

The steps in Theorem 1 are illustrated in Figure 8 for our running example. The arbitrary shift stream in Figure 8a is transformed into a prepend/shift/skip tuple in Figure 8b.

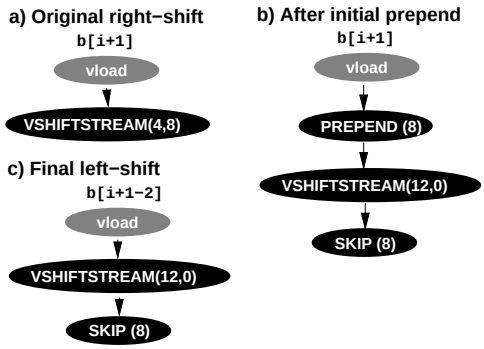

Figure 8. Transform a right-shift to left-shift.

The prepend operation is then propagated until it is processed by a memory operation, `b[i+1]` in Figure 8c. Note that, during this process, we set the free variable W associated with prepend to V.

Proof. Since `Skip` is the inverse of `Prepend`, the following $S_x \equiv \texttt{Skip}_V(\texttt{Prepend}_V(S_x, x), x)$ holds for arbitrary stream S_x and amount x. Thus we may freely set S_x to be $\text{VSHIFTSTREAM}(S, to)$ and x to be to:

$$\text{VSHIFTSTREAM}(S, to) \equiv \texttt{Skip}_V(\texttt{Prepend}_V(\text{VSHIFTSTREAM}(S, to), to), to).$$

Using definition D.3 to permute VSHIFTSTREAM and `Prepend`, we obtain,

$$\text{VSHIFTSTREAM}(S, to) \equiv \texttt{Skip}_V(\text{VSHIFTSTREAM}(\texttt{Prepend}_V(S, to), 0), to). \quad (6)$$

Parsing the left hand side of Equation (6), we note that prepending the original stream S by to bytes allows us to shift the prepended stream $\texttt{Prepend}_V(S, to)$ to offset 0. From previous work [6], we known that shifting an arbitrary stream to offset zero can always be implemented by a shift-left code sequence. Furthermore, when to bytes are skipped from the resulting shifted stream, we have the exact same stream S, as indicated by the right hand side of Equation (6).

The last statement to prove is that Step 3 does not require code. Since Step 2 always produces a stream of offset zero, Step 3 always skips to bytes from offset zero. By definition, the offset to satisfies $0 \leq to < V$. Thus, Step 3 never skips a whole vector register, only a few elements in that first register. As long as it is only a few elements in a register that need to be skipped, there is no need to issue code to erase such values; we must simply keep track of which one they are and not use them in the final store operation. □

4 Handling Length Conversion

A **length conversion** represents any operation that converts a value of one length to a value of a different length.

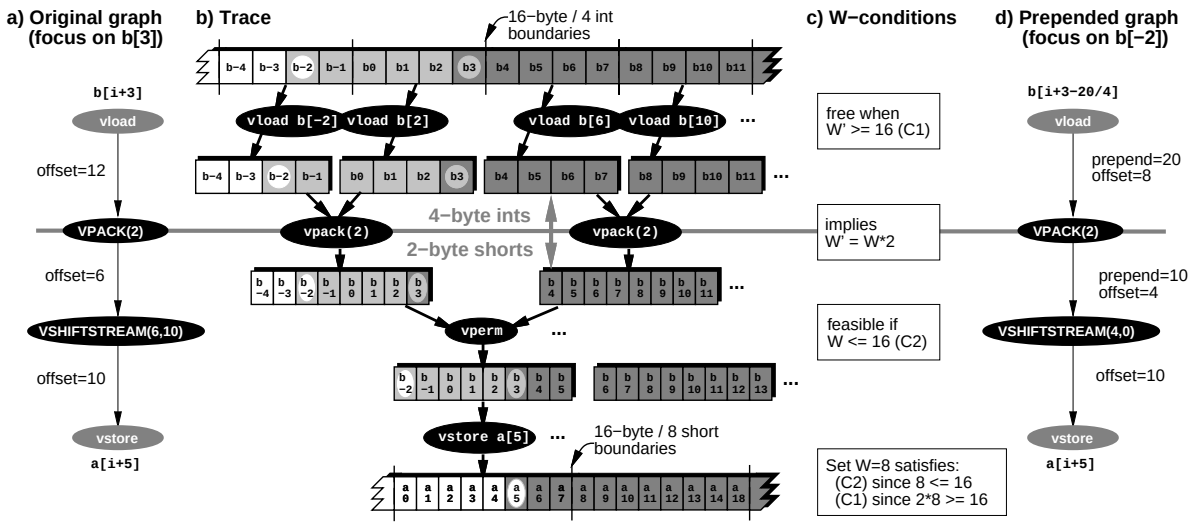

Figure 9. Conversion and shifting streams.

The most common length conversions occur in data conversions between types of different sizes[2].

4.1 Stream Pack and Unpack

We define two new stream operations to represent length conversions in our tree representation:

- VPACK(S, f). This operation packs a stream S by a factor f. For example, a conversion from 4- to 2-byte data types is a stream pack with a factor of 2.
- VUNPACK(S, f). This operation unpack stream S by a factor f. For example, conversion from 4 to 8 byte data types is a stream unpack with a factor of 2.

The uniqueness of stream pack and unpack is their ability to scale stream offsets and lengths. This can be formally expressed as,

$$\begin{aligned} \mathit{Offset}(\text{VPACK}(S,f)) &= \mathit{Offset}(S)/f \\ \mathit{Length}(\text{VPACK}(S,f)) &= \mathit{Length}(S)/f \\ \mathit{Offset}(\text{VUNPACK}(S,f)) &= \mathit{Offset}(S)*f \\ \mathit{Length}(\text{VUNPACK}(S,f)) &= \mathit{Length}(S)*f. \end{aligned}$$

Such scaling effect also applies to the implementation of prepend. Recall in Section 3 that a prepend to a register stream has to be propagated to the leaves of the expression tree until a memory stream is reached. We define the sinking of prepend past a stream pack and unpack node as follows:

$$\texttt{Prepend}_W(\text{VPACK}(S,f),x) \Rightarrow \text{VPACK}(\texttt{Prepend}_{W*f}(S, x*f), f) \quad (D.7)$$

$$\texttt{Prepend}_W(\text{VUNPACK}(S,f),x) \Rightarrow \text{VUNPACK}(\texttt{Prepend}_{W/f}(S, x/f), f). \quad (D.8)$$

where in (D.8), W and x must be divisible by f.

Note that packing and unpacking not only scales the amount being prepended, x, but also the register width associated with the prepend, W. Both scalings are necessary to ensure that, in the presence of stream pack and unpack, the properties of a prepended stream can still be computed by (4) and (5).

4.2 Problem Illustration and Overall Approach

The scaling effects of stream pack and unpack introduce two new problems to the runtime alignment handling scheme presented in Section 3. These two problems forces us to break the $W = V$ assumption made in Section 3.

1. Due to the scaling of stream offsets, stream shifts inserted by the data reorganization phase may contain runtime offsets that are greater than V. This causes problem because one cannot generate code to shift a stream by a runtime value that may be more than V.

   Note that Theorem 1 states that any stream shift VSHIFTSTREAM(S, to) can be normalized to VSHIFTSTREAM$(\texttt{Prepend}_W(S, to), 0)$. According to Equation (4), the offset of the prepended stream is $(\mathit{Offset}(S) - to) \bmod W$. Therefore, to guarantee that the normalized stream is shifted by no more than V bytes, we must instantiate W carefully so that it satisfies the $W \leq V$ condition.

2. According to the definition of prepend in D.1, the actual prepend happens at memory stream only, *i.e.*,

   $$\texttt{Prepend}_W(\text{VLOAD}(addr(i)),x) \Rightarrow \text{VLOAD}_W(addr(i) - x).$$

[2]This paper focuses on the length conversion aspect of data conversions. Further data processing beyond length conversion, such as sign extension, can be handled by regular arithmetic operations.

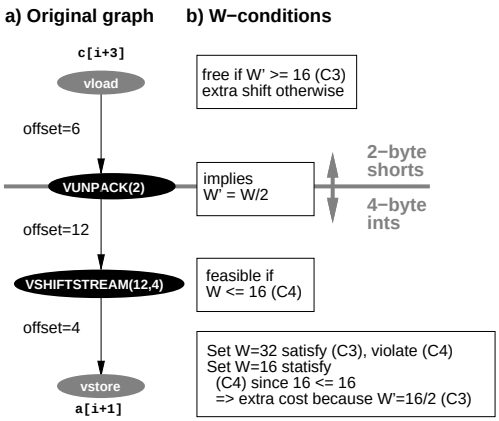

Figure 10. Prepend and unpack streams.

Recall that the address truncation at vector load is determined by the W associated with the prepend. Since W can be scaled when propagating the prepend past stream pack/unpack nodes, it can be a value different from V. This causes problems in code generation because the hardware only supports V-byte address truncation.

The solution to this problem is to perform non V-byte truncation. Let us consider the following two cases.

- When $W > V$ and W is divisable by V, then $\texttt{Prepend}_W(\text{VLOAD}(addr(i)), x)$ can be implemented as,

  $$\text{VLOAD}(addr(i) - x - (addr(i) - x) \bmod W).$$

- When $W < V$ and V is divisible by W, then $\texttt{Prepend}_W(\text{VLOAD}(addr(i)), x)$ can be implemented as,

  $$\text{VSHIFTSTREAM}(\text{VLOAD}(addr(i) - x), (addr(i) - x) \bmod W).$$

  Note that, the second case introduces an additional stream shift. Therefore, in general, we should avoid prepending a memory node with $W < V$.

The solutions to both problems are intertwined. They both rely on choosing the right W associated with the prepend during stream normalization. On the one hand, the W associated with the prepend must be *small* enough to satisfy $W \leq V$ in order to ensure a normalized stream be shifted by no more than V bytes. On the other hand, the W associated with the prepend should be *large* enough so that the W' associated with the prepend after being propagated to memory nodes satisfies $W' \geq V$ to avoid the introduction of additional stream shifts.

Consider the example of `a[i+5]=(short)b[i+3]` where `a` is an array of `short` and `b` is an array of `int`. Figures 9a and 9b illustrate the steams of a valid simdization of the loop. One can observe that the computations above the `vpack` are operating on `int` with 4 values per vector register. Below the `vpack`, however, the computations are operating on `short` with 8 values per vector register.

The conditions faced when selecting the W value associated with the shift streams are shown in Figure 9c. Starting at the shift stream, Condition C2 indicates that $W \leq 16$. Propagating W through the pack node, we get that no overhead are incurred if Condition C1 is satisfied, *i.e.*, $W' = 2W \geq 16$. In this case, selecting W as 8 satisfies both conditions. The prepend amount can now be computed, and is shown in Figure 9d.

Figure 10a illustrates an example with `a[i+1] = (int)c[i+3]` where `a` is an array of int and `c` is an array of short. The conditions faced when selecting the W values associated with the shift streams are illustrated in Figure 10b. This is a case where we cannot satisfy the conditions at the shift ($W \leq 16$) and at the load ($W' = W/2 \geq 16$) at the same time.

Note that this problem can be alleviated by a small change to the shift stream placement policy. Consider the W associated with a stream shift and a final W' after this prepend is propagated to one of its memory nodes. This W' is determined by the pack and unpack nodes traversed during prepend propagation. We can easily prove that when $W > W'$, we can not find a W that satisfies both conditions mentioned above. Fortunately, we may alter the shift placement policy to avoid such conditions, namely we avoid inserting stream shift where there is an unpack node between itself and memory node, and where there is no other stream shifts in between.

5 Putting it All Together

This section explains how to incorporate the new runtime alignment handling and length conversion handling into the three phases of the framework. The new alignment handling framework now consists of the following three phases.

Phase 1: Data reorganization is augmented from the data reorganization phase described in Section 2.2 with several additional considerations. First of all, Eager-Shift and Lazy-Shift can be applied to runtime alignment, and all shift policies can be applied to expression trees with length conversions. Secondly, when propagating stream offsets, *e.g.*, propagating the alignment of the store to the load during an Eager-Shift, the stream offset needs to be scaled up (down) by the packing factor when passing pack and unpack nodes. Thirdly, we should avoid placing a stream shift into a tree such that there is an unpack between this node and the memory node and no other stream shifts in between. Lastly, when there is a degree of freedom to place stream shift at either end of a length conversion, a more efficient policy should place the stream shift at the shorter end

PrependStream(src, to, W**)**

```
if n ≡ VLOAD(addr(i))
  return VLOAD_W(addr(i) − to)
if n ≡ VOP(src_1, ..., src_n)
  for (k = 1..n) src'_k ← PrependStream(src_k, to, W)
  return VOP(src'_1, ..., src'_n)
if n ≡ VSTREAMSHIFT(src, to)
  return VSHIFTSTREAM(PrependStream(src, to, W'), 0)
if n ≡ VPACK(src, f)
  return VPACK(PrependStream(src, to*f, W*f), f)
if n ≡ VUNPACK(src, f)
  return VUNPACK(PrependStream(src, to/f, W/f), f)
```

NormalizeShift(n**)**

```
if n ≡ VSTORE(addr(i), src)
  return VSTORE(NormalizeShift(src))
if n ≡ VLOAD(addr(i)) return VLOAD(addr(i))
if n ≡ VOP(src_1, ..., src_n)
  for (k = 1..n) src'_k ← NormalizeShift(src_k)
  return VOP(src'_1, ..., src'_n)
if n ≡ VSTREAMSHIFT(src, to)
  src' ← PrependStream(src, to, W)
  return VSHIFTSTREAM(NormalizeShift(src'), 0)
if n ≡ VPACK(src, f)
  return VPACK(NormalizeShift(src, f))
if n ≡ VUNPACK(src, f)
  return VUNPACK(NormalizeShift(src, f))
```

Figure 11. Normalization of Stream Shift.

as it takes less instructions to shift a stream with a shorter length.

Phase 2: Stream shift normalization is a new phase that "normalizes" any stream shift in the augmented tree produced by the data reorganization phase into a stream left-shift. Figure 11 gives the algorithm for normalizing stream shift in an expression tree. The W values associated with a shift stream node are selected so that they are valid ($W \leq V$ at stream shift nodes) and minimize overheads ($W' \geq V$ at memory nodes).

Phase 3: Code generation remains the same as described in Section 2.2 except that it ignores the `Skip` operation during code generation. Note that, after pushing `Prepend` towards memory streams (as shown in Figure 8c), `Prepend` is not present in the expression tree produced by the shift normalization phase. Therefore, no handling of `Prepend` is needed in this phase. In addition to that, `vpack` and `vunpack` operations need to be mapped to native permutation instructions. On AltiVec, for example, `vpack` and `vunpack` can be mapped directly to `vec_pack` and `vec_unpack`. On other platforms, these operations can be mapped to permutation instructions where the permutation mask can be computed from the packing factor[3].

[3]One can further optimize the code generation by combining pairs of consecutive pack/unpack and shift operations into a single permutation instruction.

6 Evaluation and Experimental Results

6.1 Compiler Infrastructure and Testing Platform

The proposed simdization algorithm is implemented in IBM's XL production compiler. Prior to simdization, the compiler applies high-level optimizations such as loop distribution (to split parts that cannot be simdized), if conversion (to remove control flow), and loop versioning (to create multiple versions of a loop). After simdization, it invokes further high-level optimizations such as loop fusion and loop unrolling, as well as traditional backend optimizations.

Our simdization infrastructure currently simdizes innermost loops with stride-one memory accesses as well as induction, reduction, and private variables. It performs interprocedural alignment analysis to obtain accurate alignment information. It simdizes multiple statement loops with arbitrary combinations of data types and compile-time/runtime alignments/loop bounds. It attempts to minimize redundant sign extensions to reduce unnecessary packing/unpacking overhead.

Measurements are performed on an IBM PowerPC PPC970 processor with full SIMD VMX support as found on an Apple PowerMac G5. In addition to 5 scalar units (2 fix point, 2 floating point, and 1 branch), it provides 2 SIMD units (1 arithmetic and 1 permute) and 32 additional 16-byte wide vector registers. The SIMD units support vectors of char, short, int, and float. The two memory units are shared between the scalar and vector units. In vector mode, the memory units support only 16-byte aligned memory accesses. Instructions are dispatched in groups of up to 5 instructions.

Performance is reported as a speedup factor[4] achieved by the automatically simdized loop over the sequential version of the same loop compiled at the same optimization level.

6.2 Evaluation of Alignment Overhead

We first investigate the performance impact of both compile time and runtime data misalignment on a synthesized benchmark of loops that exhibits a high ratio of misaligned data references to computations. We analyze the simdization speedup factors along three orthogonal dimensions: 3 data sizes, compile time/runtime alignments, and 3 shift placement policies. For each point, we report the harmonic mean over 50 loops with identical characteristics (2 loads, 1 add, 1 store) but different memory alignments (random, 50% bias toward one random alignment in each loop).

[4]Speedup factor is the ratio of the times for a non-simdized loop over the simdized loop. Time is measured with hardware counters and includes branching, address computation, and extra setup for simdized computation overheads. Memory effects are minimized by warming up the cache and sizing the data to fit in one L1 cache set.

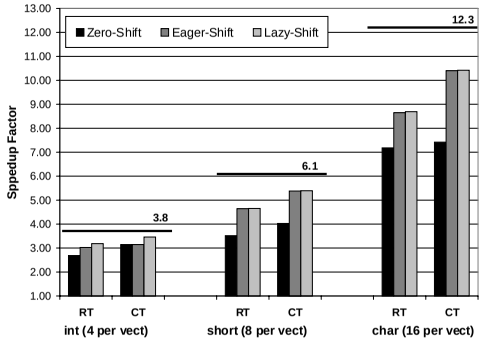

Figure 12. Simdization speedups for loops with high fraction of misaligned references.

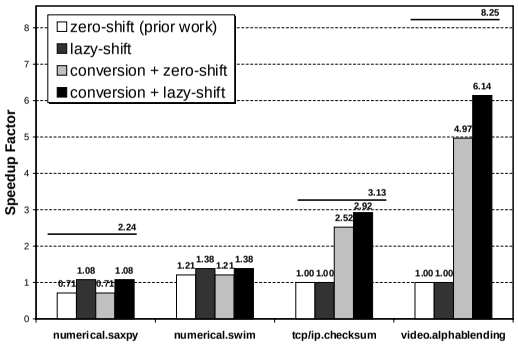

Figure 13. Simdization speedups for representative kernels.

The results in Figure 12 are first grouped in data sizes (int, short, and char). Within each size group, the measurements are then grouped into compile time alignment (CT) and runtime alignment (RT). Individual bars in a group correspond to the Zero-, Eager-, and Lazy- Shift placement policies.

For each data size in Figure 12, we report an empirical maximum speedup factor (depicted with an horizontal bold line) achieved by simdizing a loop where all accesses are aligned. We observe sub-linear speedups, which we believe are mainly due to compiler backend issues with the restricted selection of indexing/auto-incrementing addressing modes for VMX memory instructions.

Let us now focus on the fraction of the empirical maximum speedup achieved in the presence of data misalignment. This fraction is visualized as the gap between the individual vertical bars and the horizontal bold line in Figure 12. This gap is due to the overhead of shifting misaligned streams. This overhead is particularly critical on the PPC970 because, while it has two memory pipes, there is only one permute pipe for stream shifts.

With compile time alignment (CT), more aggressive shift placement policies like Lazy-Shift [6] does significantly better than the non-optimized Zero-Shift [13] policy. For example, Lazy-Shift achieves between 85 to 90% of the empirical maximum speedup over the 3 input data size.

Without compile time alignment, however, prior work (RT/Zero-Shift policy) performs poorly as it achieves only 57-70% of the empirical maximum speedup. This low speedup is due to the fact that each memory stream is realigned, due to code generation issue, even when some of them may be relatively aligned. This is not an analysis problem, as relative alignment is often trivial, *e.g.*, both `p[i]` references in `p[i]=p[i]+q[i]` are trivially relatively aligned.

Using our technique, we increase the fraction of the maximum empirical speedup from a 57-70% to a 70-83% range in the presence of runtime alignment. This corresponds to an improvement in the simdization speedup factors by 19-34% over prior art for runtime alignment.

6.3 Kernel Performance Evaluation

We report here the performance evaluation for 4 kernels extracted from two numerical applications, a video/graphic application, and a communication application. The two numerical kernels have runtime alignments, the rest have both runtime alignments and length conversion.

We report in Figure 13 the speedup factor achieved by each kernel with and without the proposed runtime alignment and conversion support proposed in this paper. The horizontal bold line above each group of bars indicates an empirical maximum speedup achieved by a similar kernel where memory accesses were forced to be aligned.

Numerical.saxpy is a numerical blas routine computing `alpha*X+Y` where `X` and `Y` are single precision vectors. The empirical maximum speedup factor of 2.24 is between the maximum floating-point speedup factor of 2 (2 scalar arithmetic units versus 4 values in 1 SIMD arithmetic unit) and memory speedup factor of 4 (2 scalar memory units versus 4 values in 2 SIMD memory units). The achieved speedups are much lower, due to a memory disambiguation problem in our backend that only impacts the scheduling of SIMD codes.

Numerical.swim is a kernel extracted from a loop accounting for 29% of the total execution time of the SPEC95 swim benchmark (code shown in Figure 3). Misaligned data accesses are runtime because of the 513 by 513 arrays dimensions (513 floats are not a multiple of 16 bytes). We could not obtain an empirical maximum speedup factor because some of the streams in the loop are relatively misaligned regardless of the dimensions of the arrays. The speedup factor increases from 1.21 to 1.38 thanks to a reductions in the number of stream shifts from 26 to 18.

Tcp/ip.checksum is a communication routine that verifies

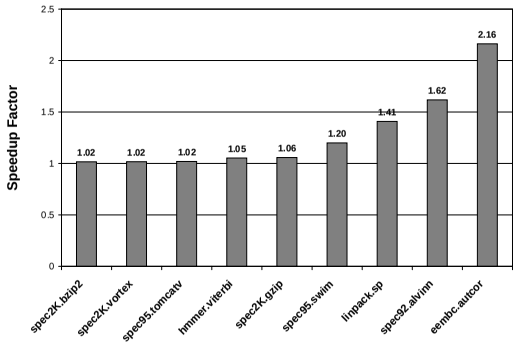

Figure 14. Simdization speedups for benchmarks.

the integrity of incoming network packets. It accumulates an integer checksum from the short values of 4K char buffer passed in as a pointer. The maximum empirical speedup is 3.13, between the maximum computational speedup of 2 (for integers) and memory speedup factor of 8 (for shorts), because of the reduction overhead. To simdize the loop, we must turn on our support for conversion, which gets a 2.52 speedup with Zero-Shift. The speedup further increases to 2.92 since our technique also enables the optimized data reorganization (Lazy-Shift).

Video.alphablending is a graphics routine that blends pictures with variable amount of transparency. Its memory streams are chars and some of the processing is performed as shorts. It thus requires conversion from char to short, and back. The empirical maximum speedup factor of 8.25 for all aligned case is between the theoretical computational speedup factor of 4 (for shorts) and memory speedup factors of 16 (for chars). To simdize the loop, we must turn on our support for conversion, which gets a 4.97 speedup with Zero-Shift. The speedup further increases to 6.14 since our technique also enables the Lazy-Shift policy.

6.4 Application Results

We report in Figure 14 the speedup factors (ranging from 1.02 to 2.16) achieved by simdizing 9 benchmark applications. Three of SpecFP programs (swim, tomcatv, and alvinn) are manually converted to use single-floating point.

In general, the speedup factor are more modest, as only part of applications are suitable for simdization. In addition, 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.

Bzip and **vortex** achieve minor speedups because the simdized loops do not significantly contribute to the overall computation. Three of the simdized loops from **bzip** contain runtime alignment and conversion. **Tomcatv** achieves moderate performance improvement. The hot loops in **tomcatv** use 513 by 513 arrays, making such references runtime. 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 as scalar ones, thus resulting in spurious loads and data reorganizations. Finally, only a few reduction loops from **viterbi** are simdized.

The next benchmarks indicate more promising speedup factors. In **gzip**, we successfully simdize a all aligned loop performing saturating arithmetic. In **swim**, we simdize one of the three hot loops where all array accesses have runtime alignments. The other two loops cannot be simdized because of the use of divide operation and dependences. We get a speedup factor of 1.41 in **linpack** in single precision mode. In **alvinn**, one of the hot loops we simdize contains reduction with runtime alignments. In **autocor**, a 2.16 speedup is achieved by successfully simdizing the loop (as shown in Figure 4) that involves runtime alignment, length conversion, and reduction.

7 Related Work

There has been a recent upsurge of compiler research on automatic vectorization for SIMD units [3, 2, 4, 7, 8, 11]. Two principal techniques have been used, the traditional loop-based vectorization pioneered for vector supercomputers [1, 15] and the unroll-and-pack approach first proposed by [8]. Our simdization scheme falls into the first category among others [2, 11, 14, 10].

As being covered in detail in Section 2 and 3, the work that is closely related to ours is done by [6], which also lays the foundation of this work. Our work extends [6] in two directions. First, our technique enables Eager and Lazy shift policies on runtime alignment handling, whereas [6] allows only zero shift policy for runtime alignment. Secondly, our technique enables alignment handling in the presence of length conversion operations.

Besides [6], the most extensive alignment handling was implemented in the VAST compiler [13]. Like [6], VAST handles loops with multiple misaligned references, unknown loop bounds, runtime alignments, and conversions. However, VAST supports only one shift policy, *i.e.*, the Zero-Shift policy. Therefore, our scheme has the advantage of exploiting the benefit of the other three additional placement policies.

The most extensive discussion of alignment considerations is in [9]. However, [9] is concerned with the detection of memory alignments and with techniques to increase the number of aligned references in a loop, whereas our work focuses on generating optimized SIMD codes in the presence of misaligned references. The two approaches are complementary. The use of loop peeling to align accesses

was discussed in [9, 2]. The loop peeling scheme is equivalent to the Eager-Shift policy with the restriction that all memory references in the loop must have the same misalignment.

Direct code generation for misaligned references has been discussed by several prior works in the context of VIS [4] and SSE2 [2]. Their methods are equivalent to Zero-Shift but are not discussed in the context of general misalignment handling. Furthermore, neither [4] nor [2] exploit the reuse when aligning a stream of contiguous memory. As shown in [6], lack of exploiting the reuse may result in a performance slowdown of more than a factor of 2.

8 Conclusion

This paper addresses two important limitations of the alignment framework proposed by [6], *i.e.*, inefficient handling of runtime alignment and a lack of support for length conversion.

In this paper, we propose a novel technique to efficiently shift arbitrary streams to an arbitrary offset, regardless whether the alignments or offsets are known at the compile time or not. This technique enables the application of the more advanced alignment optimizations such as Eager- and Lazy-Shift policies to runtime alignment. This enablement has a significant impact on runtime alignment performance. On a G5 machine with a 16-byte wide VMX/AltiVec unit, our technique demonstrates a 19% - 34% improvement of performance over prior art on a benchmark stressing the impact of misaligned data.

We address the second limitation by supporting length conversion in alignment handling. In this paper, length conversion is discussed in the context of data conversions. However, our approach is general enough to abstract the length conversion nature of any operation that consumes streams of one length and produces a stream of a different length. Putting it all together, we demonstrated speedup factors from 1.02 to 8.14 for real benchmarks over sequential execution.

References

[1] J. R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, (4):491–542, October 1987.

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

[3] A. J. Bik. *The Software Vectorization Handbook*. Intel Press, 2004.

[4] G. Cheong and M. S. Lam. An Optimizer for Multimedia Instruction Sets. In *Second SUIF Compiler Workshop*, August 1997.

[5] M. Corporation. AltiVec Technology Programming Interface Manual, June 1999.

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

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

[8] S. Larsen and S. 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.

[9] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of 11th International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

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

[11] D. Naishlos, M. Biberstein, S. Ben-David, and A. 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.

[12] G. Ren, P. Wu, and D. Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.

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

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

[15] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.