Vectorization for SIMD Architectures with Alignment Constraints

Alexandre E. Eichenberger Peng Wu Kevin O'Brien

IBM T.J. Watson Research Center
Yorktown Heights, NY

{alexe,pengwu,caomhin}@us.ibm.com

ABSTRACT

When vectorizing for SIMD architectures that are commonly employed by today's multimedia extensions, one of the new challenges that arise is the handling of memory alignment. Prior research has focused primarily on vectorizing loops where all memory references are properly aligned. An important aspect of this problem, namely, how to vectorize misaligned memory references, still remains unaddressed.

This paper presents a compilation scheme that systematically vectorizes loops in the presence of misaligned memory references. The core of our technique is to automatically reorganize data in registers to satisfy the alignment requirement imposed by the hardware. To reduce the data reorganization overhead, we propose several techniques to minimize the number of data reorganization operations generated. During the code generation, our algorithm also exploits temporal reuse when aligning references that access contiguous memory across loop iterations. Our code generation scheme guarantees to never load the same data associated with a single static access twice. Experimental results indicate near peak speedup factors, *e.g.*, 3.71 for 4 data per vector and 6.06 for 8 data per vector, respectively, for a set of loops where 75% or more of the static memory references are misaligned.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors - *compilers, code generation, optimization*

General Terms

Languages, Performance, Design, Algorithms

Keywords

SIMD, compiler, vectorization, simdization, multimedia extensions, alignment

PLDI'04, June 9–11, 2004, Washington, DC, USA.

1. INTRODUCTION

Multimedia extensions have become one of the most popular additions to general-purpose microprocessors. Existing multimedia extensions can be characterized as **S**ingle **I**nstruction **M**ultiple **D**ata (SIMD) units that support packed fixed-length vectors. The traditional programming model for multimedia extensions has been explicit vector programming using either (in-line) assembly or intrinsic functions embedded in a high-level programming language. Explicit vector programming is time consuming and error-prone. A promising alternative is to exploit vectorization technology to automatically generate SIMD codes from programs written in standard high-level languages.

Although vectorization has been studied extensively for traditional vector processors decades ago, vectorization for SIMD architectures has raised new issues due to several fundamental differences between the two architectures [1]. To distinguish between the two types of vectorization, we refer to the latter as **simdization**. One such fundamental difference comes from the memory unit. The memory unit of a typical SIMD processor bears more resemblance to that of a wide scalar processor than to that of a traditional vector processor. In AltiVec [2], for example, a load instruction loads 16-byte contiguous memory from 16-byte aligned memory, ignoring the last 4 bits of the memory address in the instruction. The same applies to store instructions. In this paper, **architectures with alignment constraints** refer to machines that support only loads and stores of register-length aligned memory.

The alignment constraints of SIMD memory units present a great challenge to automatic simdization. Consider the code fragment in Figure 1 where integer arrays `a`, `b`, and `c` are aligned[1]. Although this loop is easily vectorizable for

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

Figure 1: A loop with misaligned references.

traditional vector processors, it is non-trivial to simdize it for SIMD architectures with alignment constraints. The most commonly used policy today is to simdize a loop only if all memory references in the loop are aligned. In the presence of misaligned references, one common technique is to peel

[1]An aligned reference means that the desired data reside at an address that is a multiple of the vector register size.

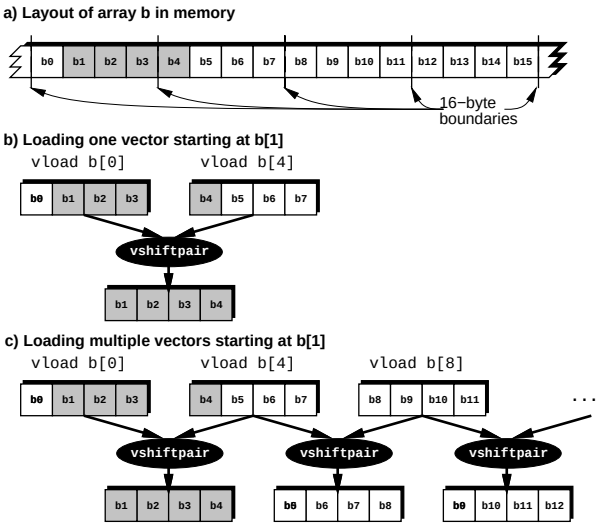

Figure 2: Loading from misaligned addresses.

the loop until all memory references inside the loop become aligned [3, 4]. The peeling amount can be determined either at compile-time or at runtime. However, this approach will not simdize the loop in Figure 1 since any peeling scheme can only make at most one reference in the loop aligned.

In this paper, we propose a systematic approach to simdizing loops with misaligned stride-one memory references for SIMD architectures with alignment constraints. This is achieved by automatically generating data reorganization instructions during the simdization to align data in registers. Using the array reference `b[i+1]` as an example, Figure 2b illustrates the basic mechanism to implement a misaligned load on a typical SIMD unit with alignment constraints. We use here instruction `vload` to load a vector from 16-byte aligned memory and instruction `vshiftpair` to select consecutive elements of two vector registers to an output vector register. The misalignment handling can be further improved, by reusing the vector loads across loop iterations as shown in Figure 2c. Our algorithm is able to exploit such reuse and guarantees that data associated with a single static reference in the original loop will not be loaded twice.

We adopt a systematic and non-intrusive approach to the handling of misalignment in simdization. First, the loop is simdized as if for a machine with no alignment constraints. Second, data reorganization operations are inserted into the simdized code to satisfy the actual alignment constraints. The second step is our key contribution to alignment handling and is the primary focus of this paper.

The second step occurs in following two phases which communicate via a **data reorganization graph**.

Data Reorganization Phase inserts data reordering operations in the code produced by the previous step to satisfy the actual alignment constraints. Optimizations are applied to minimize the number of data reordering operations generated. This phase is relatively architecture independent and its output is a data reorganization graph.

SIMD Code Generation Phase maps the simdized operations (including the data reordering operations inserted by the previous phase) to SIMD instructions specific to the target platform. This phase addresses issues such as runtime alignments, unknown loop bounds, multiple misalignments, and multiple statements.

Performance evaluation indicates that near peak speedup can be achieved even in the presence of large numbers of misaligned references in the loop. Comparing the dynamic instruction count of simdized codes generated by our production compiler to an ideal scalar instruction count, we achieve the following speedups over a wide range of loop benchmarks. With 4 integers packed in a vector register and with on average 3/4 of the static memory references misaligned, speedups of up to 3.71 and 2.69 are achieved with and without static alignment information, respectively. With 8 short integers packed in a vector register and with on average 7/8 of the memory references misaligned, we achieve speedups of up to 6.06 and 4.64 with and without static alignment information, respectively.

In summary, this paper makes the following contributions:

- Introduces a new abstraction, the data reorganization graph, to incorporate alignment constraints and to enable the systematic generation and optimization of data reorganization operations during simdization.
- Proposes a robust algorithm to simdize misaligned loads and stores including loops with multiple misaligned references, runtime alignments, unknown loop bounds, and multiple statements.
- Proposes an efficient code generation algorithm that exploits reuse on stride-one misaligned memory references to minimize the number of vector loads and stores.
- Demonstrates near peak speedup even in the presence of large numbers of misaligned memory references.

The rest of the paper is organized as follows. Section 2 gives an overview on SIMD architectures. We then describe the data reorganization and code generation phases of the simdization algorithm in Sections 3 and 4, respectively. Experimental results are presented in Section 5. Section 6 discusses the related work and we conclude in Section 7.

2. BACKGROUND

2.1 SIMD Architectures

Multimedia extensions have been adopted by most major computer manufactures such as MMX/SSE for Intel, 3DNow! for AMD, VMX/AltiVec for IBM/Motorola, and VIS for SUN. Similar architectures can also be found in graphics engines and game consoles such as NVDIA, ATI, PS2, and XBOX, and some of the DSP processors.

These processing units can be characterized as SIMD processors operating on packed fixed-length vectors. A typical SIMD unit provides a set of vector registers that are usually 8- or 16-byte wide. It supports SIMD operations on 1, 2, 4, and possibly 8 byte data types. For example, a 2-byte vector add on a 16-byte vector would add 8 distinct data in a vector in parallel. In terms of memory units, most media processing units mentioned before[2] provide a load-store unit similar to AltiVec's as described in Section 1.

[2]SSE2 supports some limited form of misaligned memory accesses which incurs additional overhead.

2.2 Generic Data Reorganization Operations

Most SIMD architectures support a rich set of operations to reorder data in vector registers. These operations are heavily used in our alignment handling. In order to present a general-purpose simdization scheme, we introduce three generic data reorganization operations. These generic operations can be easily mapped to instructions of specific platforms. We will illustrate their implementations on existing SIMD architectures using AltiVec as an example. In the rest of the section, we use V to denote the vector length.

`vsplat`(x) replicates a scalar value x to form a full vector by V/`sizeof`(x) times.

This operation is directly supported by most SIMD architectures, *e.g.*, `vec_splat` on AltiVec.

`vshiftpair`(v_1, v_2, ℓ) selects bytes $\ell, \ell+1, \ldots, \ell+V-1$ from a double-length vector constructed by concatenating vectors v_1 and v_2, where $0 \leq \ell < V$.

This operation can be implemented by permute operations that combine two vectors through a permute vector, *e.g.*, `vec_perm` on AltiVec. Each byte of the permute vector specifies which byte of the two concatenated input vectors is selected. The permute vector can be constructed as vector literal $(\ell, \ell+1, \ldots, \ell+V-1)$ if ℓ is known at compile-time, or as the result of adding `vsplat`((`char`)ℓ) with vector literal $(0, \ldots, V-1)$.

`vsplice`(v_1, v_2, ℓ) splices two vectors v_1 and v_2 at a splice point specified by an integer value ℓ. Specifically, it concatenates the first ℓ bytes of v_1 with the last $(V-\ell)$ bytes of v_2 when $0 < \ell < V$, copies v_1 when $\ell \leq 0$, and copies v_2 when $\ell \geq V$.

This operation can be implemented by the select operation available on most SIMD architectures, *e.g.*, `vec_sel` on AltiVec. For each bit of the output vector, this operation selects the bit from one of the two input vector registers based on a mask vector. The mask vector can be computed as the result of comparing vector literal $(0, 1, \ldots, V-1)$ against `vsplat`((`char`)ℓ).

3. DATA REORGANIZATION PHASE

In this section, we first give an intuitive example of why byte reordering operations are needed for alignment handling in Section 3.1. We then introduce the concept of stream and stream shift in Section 3.2 and the data reorganization graph in Section 3.3. Graph optimizations to minimize the amount of data reorganization are presented in Section 3.4.

For the simplicity of the description, the code examples used in this section assume that the vector length is 16 bytes, the base address of an array is 16-byte aligned, and the values are 32 bit integer values.

3.1 Constraints of a Valid Simdization

Consider our original example of `a[i+3]=b[i+1]+c[i+2]` in Figure 1. Since there is no loop-carried dependence, the loop can be easily simdized for machines with no alignment constraints. However, such simdized code is invalid for SIMD units that support only aligned loads and stores. Figure 3 illustrates the execution of the simdized loop on a hardware with alignment constraints.

Consider the `i=0` iteration of the simdized loop in Figure 3a, focusing on the values of expression `a[3]=b[1]+c[2]` which are highlighted by white circles on gray background

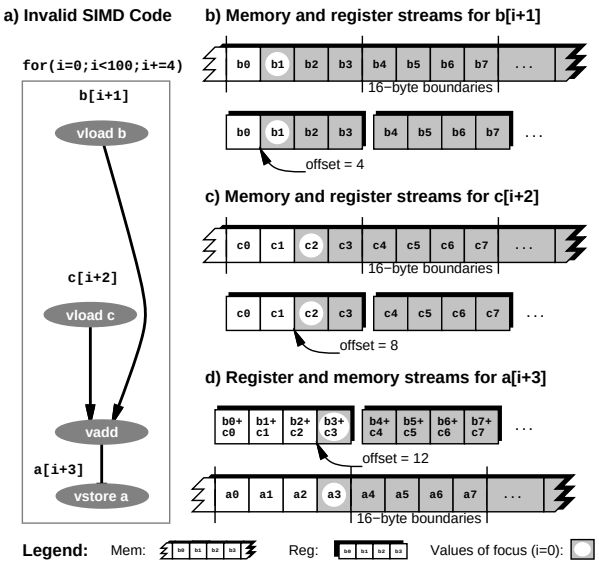

Figure 3: An invalid simdization on hardware with alignment constraints.

in Figures 3b-d. The `vload b[1]` operation loads vector `b[0]`,...,`b[3]` with the desired `b[1]` value at byte-offset 4 in its vector register, as shown in Figure 3b. Similarly, the `vload c[2]` operation loads `c[0]`,...,`c[3]` with `c[2]` at byte offset 8, as depicted in Figure 3c. Adding these two vector registers yields the values `b[0]+c[0]`,...,`b[3]+c[3]`, as illustrated in Figure 3d. This is clearly *not* the result specified by the original `b[i+1]+c[i+2]` computation.

Based on these observations, we list the following constraints that a valid simdization must satisfy:

1. When performing a vector load, the 16-byte alignment of the load address dictates the byte-offset of the data in its destination vector register. For example, the 16-byte alignment of `b[1]` and `c[2]` in memory is 4 and 8 bytes, respectively, as is the byte offset in their respective vector registers.
2. When computing vector operations (possibly excluding data reordering operations), the data involved in the original operation must reside at the same byte-offset in their respective vector registers.
3. When performing a vector store, the byte-offset of the data in the vector register must match the memory alignment of the store address. For example, `b[1]+c[2]`, when being stored to `a[3]`, must reside at byte-offset 12 in its vector register to match the memory alignment of `a[3]`.

Thus, data reorganization for a valid simdization can be summarized as reordering data in vector registers so that the above specified constraints are satisfied. The formalization of these constraints will be presented later in Section 3.3.

3.2 Streams and Stream Shifts

Given a *stride-one* memory reference in a loop, a **memory stream** corresponds to all the contiguous locations in memory addressed by that memory reference over the lifetime of the loop. For example, the gray boxes in the contiguous band in Figure 4b depict the memory stream associated with

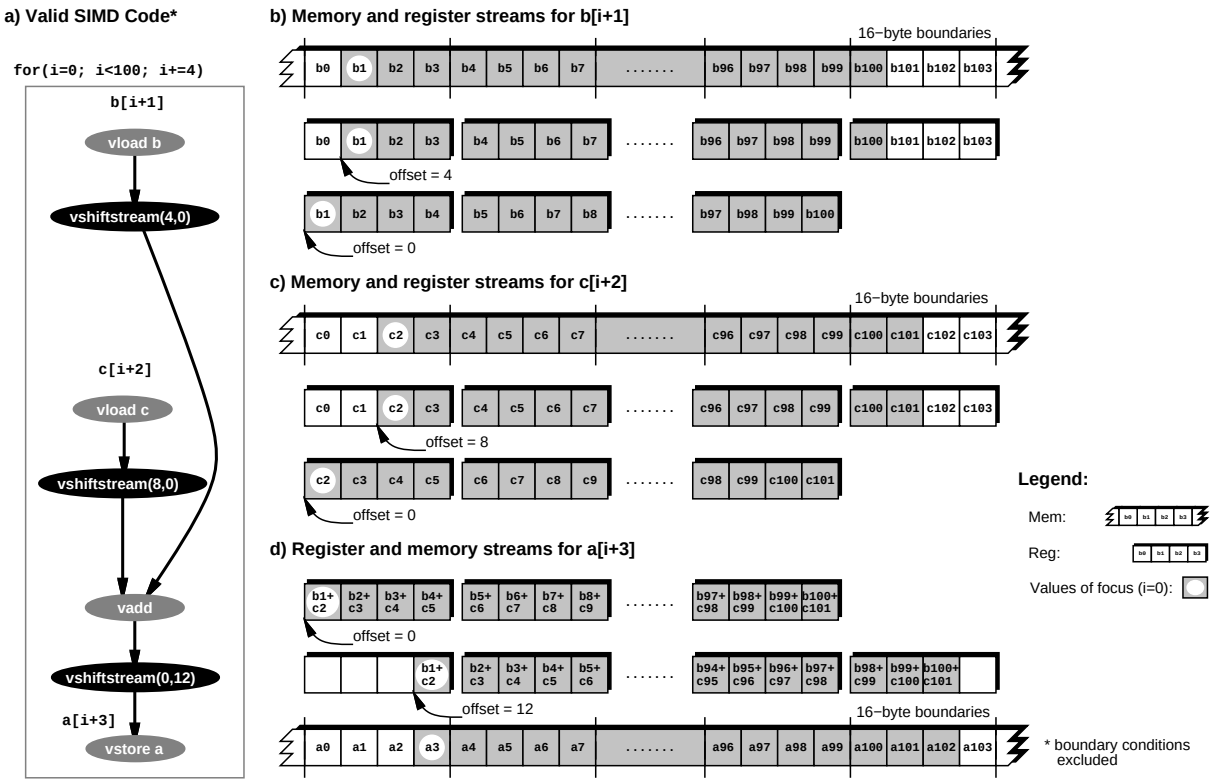

Figure 4: A valid simdization for hardwares with alignment constraints.

`b[i+1]` in the `i=0` to `99` loop, spanning the values from `b[1]` to `b[100]`. Similarly in Figure 4c, the memory stream associated with `c[i+2]` spans the values from `c[2]` to `c[101]`.

Similarly, a **register stream** corresponds to all the consecutive registers produced by a single vector operation over the lifetime of a loop. Note that, as a memory stream is read from memory by vector loads in discrete chunks of 16 bytes, extra values may be introduced at the beginning and the end of a register stream. For example, in Figure 4b, the first value in the register stream is not `b[1]` but `b[0]`.

To distinguish the desired values from the extra values in a register stream, we introduce the concept of a **stream offset**, defined as the byte-offset of the first desired value of a register stream. Namely, stream offset is the byte-offset of the data associated with the `i=0` computation. Stream offset values are by definition nonnegative and smaller than the vector length.

In Section 3.1, we establish that a simdization is valid when all of the data processed by an original operation reside at the same byte-offset in their respective vector registers. To that effect, we introduce a new data reorganization operator, `vshiftstream`(c_1, c_2), which shifts all values of a register stream among consecutive registers of that stream. Essentially, it takes an input register stream whose offset is c_1 and generates a register stream of the same values but with a stream offset of c_2.

For example, `vshiftstream(4,0)` in Figure 4a shifts the register stream associated with `vload b[i+1]` to the left by 4 bytes, as shown in Figure 4b, eliminating the extra initial value `b[0]` from the register stream. The same operator can also be used to shift values to the right, as shown in Figure 4d, where `vshiftstream(0,12)` shifts right the register stream of `b[i+1]+c[i+2]` by 12 bytes. The resulting register stream has an offset of 12, which matches the alignment of the memory stream generated by reference `a[i+3]`.

3.3 Data Reorganization Graph

A data reorganization graph is an expression tree augmented with data reordering operations. Figure 4a is an example of such a graph. Each node in the graph is associated with a stream offset property. Since the stream offset property is key to our definition of a valid data reorganization graph, we describe below how to compute the stream offset for each type of data reorganization graph nodes.

The rest of the section uses the following notations: V for the vector length, i for the loop counter, O and O_x for the stream offset associated with the current node in consideration and any other node x, respectively. For an offset known at compile times, O_x is a compile time constant that is directly used by our algorithms; otherwise, for runtime offsets, O_x is a register value that is computed at runtime by anding memory addresses with literal $V - 1$.

VLOAD($addr(i)$) This node loads a vector from a stride-one memory reference $addr(i)$. This operation produces a register stream whose stream offset is defined by the alignment of $addr(i)$, *i.e.*,

$$O \leftarrow addr(i = 0) \bmod V \tag{1}$$

VSTORE($addr(i), src$) This node stores a vector stream produced by node src to a stride-one reference $addr(i)$. This node does not have a stream offset. However, in order for the store to be valid, the stream offset of node src

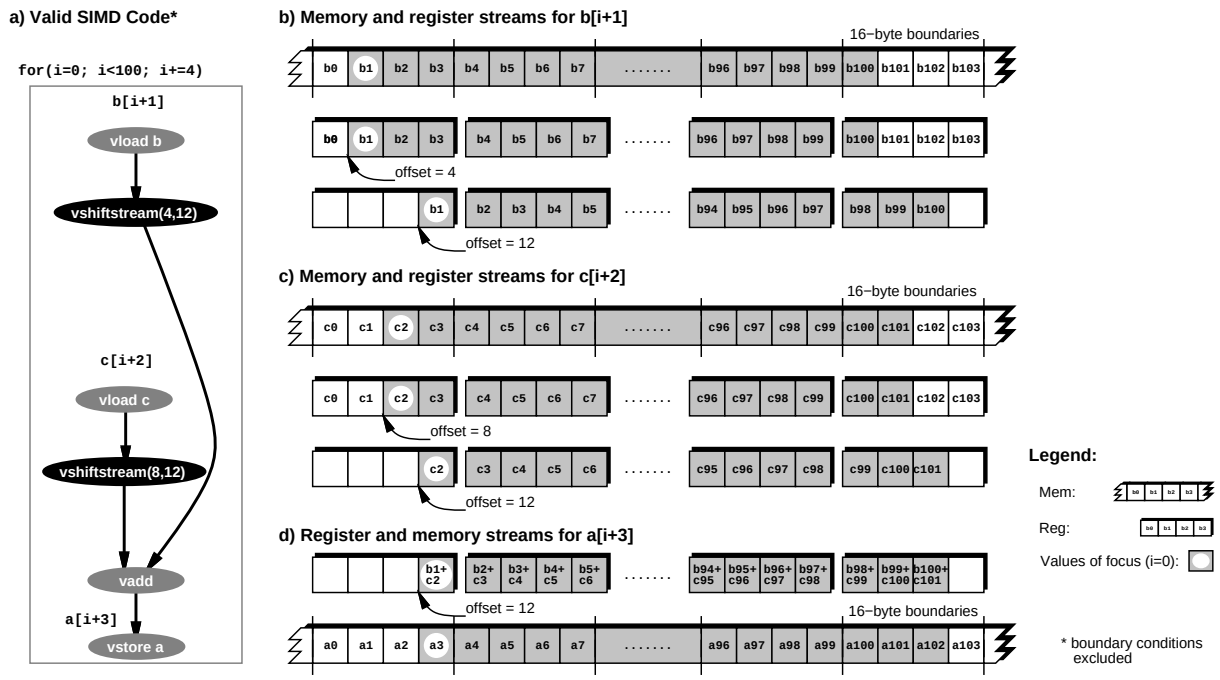

Figure 5: Illustration of the eager-shift policy for the example in Figure 4.

must satisfy the following condition:

$$O_{src} = addr(i = 0) \bmod V \qquad (C.2)$$

VOP$(src_1, \ldots, src_n)$ This node represents a regular vector operation that takes as input register streams associated with nodes $src_1, \ldots, src_n$ and produces one output register stream. In order for the computation to be valid, input register streams must have matching stream offsets, *i.e.*,

$$O_{src_1} = O_{src_2} = \ldots = O_{src_n} \qquad (C.3)$$

The stream offset of this node is defined by the uniform stream offset of its input nodes, *i.e.*,

$$O \leftarrow O_{src_1} \qquad (4)$$

VSHIFTSTREAM(src, O_{src}, c) This node shifts the register stream associated with the input node src and stream offset O_{src} to a register stream with a stream offset c. This is a data reorganization node which can change the offset of a register stream. By the definition of the operation, the stream offset of this node is:

$$O \leftarrow c \qquad (5)$$

where $0 \leq c < V$ and must be a loop invariant.

VSPLAT(x) This node replicates a loop invariant x to produce a register stream with concatenated values of x. The stream offset of this node is "undefined" and is associated with the symbol $\top$, as the same value is replicated in all register slots,*i.e.*,

$$O \leftarrow \top \qquad (6)$$

Note that $\top$ can be any defined value in $(C.2)$ and $(C.3)$.

Essentially, $(C.2)$ and $(C.3)$ specify the constraints that must be satisfied to produce a valid data reorganization graph. They are the formalization of the second and the third constraints described in Section 3.1.

3.4 Generating a Data Reorganization Graph

A valid data reorganization graph require the stream offset of each node in the graph satisfy Constraints $(C.2)$ and $(C.3)$. In the presence of misalignments, this property is only achievable by judicious placement of data reordering nodes such as VSHIFTSTREAM to the original expression tree. We investigate several policies to place vshiftstream-nodes to generate a valid data reorganization graph.

Zero-Shift Policy

The main idea behind this policy is to (1) shift each misaligned register stream to a stream offset of 0 immediately after it is loaded from memory, and (2) to shift each register stream to the alignment of the store address just before it is stored to memory. More specifically,

- For each vload-node x, insert VSHIFTSTREAM$(x, O_x, 0)$ between x and its output nodes.
- For each vstore-node x of VSTORE$(addr(i), src)$, insert VSHIFTSTREAM(src, O_{src}, c) between nodes src and x where c is equal to $addr(i = 0) \bmod V$.
- For each loop invariant node x used as a register stream, insert VSPLAT(x) between x and its output node.

The simdization example in Figure 4 uses the zero-shift policy. This policy is the least optimized in terms of the number of data reorganization operations since it inserts one VSHIFTSTREAM for each misaligned memory stream.

Eager-Shift Policy

This policy shifts each misaligned load stream directly to the alignment of the store, rather than to 0 as the zero-shift policy does. Specifically, for each vload-node x in the graph,

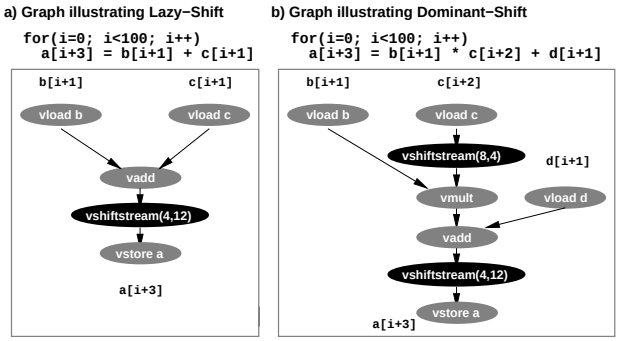

Figure 6: Lazy-shift and dominant-shift policies.

the policy inserts a VSHIFTSTREAM(x, O_x, c) between x and its output nodes, where c is the alignment of the store.

Due to code generation issues investigated in Section 4.2, this policy requires alignments of loads and stores to be known at compile-time. Figure 5 illustrates the eager-shift placement policy, lowering the total number of stream shift operations from 3 to 2.

Lazy-Shift Policy

This policy is based on the eager-shift policy but is improved further by delaying stream shifts as long as Constraints $(C.2)$ and $(C.3)$ are satisfied.

Consider the example `a[i+3]=b[i+1]+c[i+1]` in Figure 6a. Zero-shift policy would insert 3 VSHIFTSTREAM operations. Eager-shift would require 2, one for each misaligned load. This policy, however, exploits the fact that `b[i+1]` and `c[i+1]` are relatively aligned, thus satisfying $(C.3)$ and can be safely operated on as is. Only the result of the add needs to be shifted so as to match the alignment of the store, as shown in Figure 6a.

Dominant-Shift Policy

This policy further reduces the number of stream shifts by shifting register streams to the most dominant stream offset in the graph. This policy is most effective if applied after the lazy-shift policy. For example, in Figure 6b, the dominant offset is stream offset 4. Shifting stream to this offset decreases the number of VSHIFTSTREAM operations from 4 (for the zero-shift policy) to 2.

4. SIMD CODE GENERATION

This section presents the code generation algorithm. We start with an algorithm that simdizes a single-statement loop with compile-time alignments and loop bounds. We then augment the algorithm to handle multiple-statement loops, runtime alignments, and unknown loop bounds.

4.1 Assumptions and Notations

In our algorithm, we assume the loop to be simdized is an innermost loop that satisfies the following conditions:

- All memory references are either loop invariant or stride-one array references.
- The base address of an array is naturally aligned to the data length of its array elements.
- The loop counter can only appear in the address computation of stride-one references.
- All memory references access data of the same length. There is no conversion between data of different lengths.

The rest of the paper uses the following notations: V for the vector length, D for the uniform data length of all memory references in the loop, and O_x for the stream offset of a graph node x. We also denote the blocking factor of the simdized loop as B, which is computed as the number of data per vector, *i.e.*,

$$B = V/D. \tag{7}$$

4.2 Single-Statement Algorithm

This algorithm simdizes a single-statement loop where memory alignments and loop bounds are known at compile-time. We assume that the loop is normalized and has a loop counter i and an upper bound ub.

The input to the algorithm is the data reorganization graph of the single statement in the loop. The algorithm traverses the graph in postorder starting from the store-node and recursively processes each child before itself. The code generation algorithm relies on native vector operations of the target machine plus an implementation of the generic data reordering operations presented in Section 2.2, namely, `vsplat`, `vshiftpair`, and `vsplice`. In addition, we use the following helper functions:

`Runtime`(c) determines whether c is a compile-time or runtime value.

`Substitute`$(n, x \leftarrow y)$ traverses the tree starting at node n and replaces all occurrences of x by y.

`GenStoreStmt`$(addr, expr, ptr)$ generates a store statement of expression $expr$ to address $addr$ at the insertion point specified by ptr. If $addr$ is given as a string, *e.g.*, `'t'`, it represents the address of a local variable named `'t'`.

Simdizing an Expression

The following tasks are performed when processing nodes in the data reorganization graph of an expression, including all but the final store node in the graph. Store nodes are special cases that are presented in the next subsection. The detailed algorithm is given in Figure 7.

VLOAD$(addr(i))$ When processing this node, we emit a `vload` vector operation of address $addr(i)$ without further simdizing the load's inputs.

VOP$(src_1, \ldots, src_n)$ When processing this node, we first generate SIMD codes to compute every source value, which is then used by a SIMD version of the `vop` operation.

VSPLAT(x) When processing this node, we first generate traditional code to compute the x value, which is then used by a `vsplat` vector operation.

VSHIFTSTREAM(src, O_{src}, c) When processing this node, the algorithm first determines whether the register stream associated with src is shifted left (*e.g.*, Figure 4b) or shifted right (*e.g.*, Figure 4d).

When shifting a register stream left, *i.e.*, $O_{src} > c$, data from the *next* register of the src register stream is shifted into the *current* register of the stream. Consider the VSHIFTSTREAM$(\texttt{b[i+1]}, 4, 0)$ in Figure 4b. Data `b[4]` from the second register of the stream is shifted into the first register of the stream to produce `(b[1],b[2],b[3],b[4])` as the first register of the output register stream.

GenSimdExpr(n)

1 if $n \equiv$ VLOAD($addr(i)$) return `vload`($addr(i)$)
2 if $n \equiv$ VSPLAT(x) return `vsplat`(`GenExpr`(x))
3 if $n \equiv$ VSHIFTSTREAM(src, O_{src}, c)
4 return `GenSimdShiftStream`(src, O_{src}, c)
5 if $n \equiv$ VOP($src_1, \ldots, src_n$)
6 for ($k = 1..n$) $vreg_k \leftarrow$`GenSimdExpr`(src_k)
7 return `vop`($vreg_1, \ldots, vreg_n$)

GenSimdShiftStream(n, from, to)

8 $shift \leftarrow (from - to) \mod V$
9 if ($from > to$ | `Runtime`(to)) /* shift left */
10 $curr \leftarrow$ `GenSimdExpr`(n)
11 $next \leftarrow$ `GenSimdExpr`(`Substitute`($n, i \leftarrow i + B$))
12 return `vshiftpair`($curr, next, shift$)
13 else if ($from < to$ | `Runtime`($from$)) /* shift right */
14 $curr \leftarrow$ `GenSimdExpr`(n)
15 $prev \leftarrow$ `GenSimdExpr`(`Substitute`($n, i \leftarrow i - B$))
16 return `vshiftpair`($prev, curr, shift$)

Figure 7: SIMD code generation for expressions.

Since all memory streams are based on stride-one memory references, the *next* register in a register stream corresponds to the vector produced by the *next* simdized iteration. Thus, it can be computed by replacing i with $(i + B)$ in the simdized node.

When shifting a stream right, *i.e.*, $O_{src} < c$, the resulting vector register is similarly obtained by combining the *previous* and the *current* vector registers of the *src* register stream.

Simdizing a Statement

When simdizing a VSTORE($addr(i)$), extra precaution must be taken for the first and last few iterations of the original loop. Consider, for example, the store `a[i+3] = ...` originally illustrated in Figure 4d. Since `a[i+3]` has an offset of 12 bytes, only 4 bytes of the newly computed data should be stored during the first iteration of the simdized loop. Similarly, only 12 bytes of the newly computed data should be stored in the last iteration of the simdized loop.

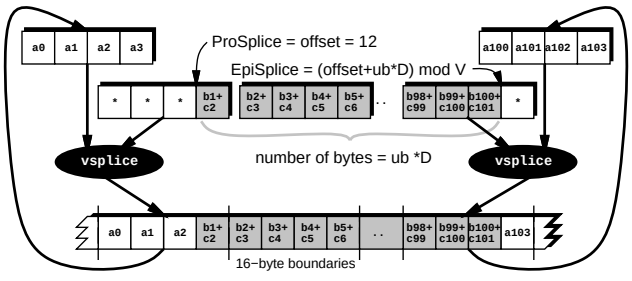

Figure 8: Special cases for prologue and epilogue.

In general, such partial vector stores can only occur in the first and/or the last iterations of a simdized loop. To handle such cases without impact on the steady state performance of a loop, we peel the first and the last iteration of a simdized loop into, respectively a prologue and epilogue that are customized to handle partial stores. As illustrated in Figure 8, on SIMD units without dedicated hardware support, partial stores are implemented by loading the original value prior to the store, splicing it with the newly computed value, then storing the spliced value back into memory using `vsplice` operation. The algorithm to handle the prologue, steady-state, and epilogue is given in Figure 9.

GenSimdStmt-Prologue(addr(i), src)

1 $splicePoint \leftarrow addr(0) \mod V$
2 $new \leftarrow$ `GenSimdExpr`(src)
3 $old \leftarrow$ `vload`($addr(0)$)
4 $spliced \leftarrow$ `vsplice`($old, new, splicePoint$)
5 `GenStoreStmt`($addr(0), spliced,$ '`in prologue`')

GenSimdStmt-Steady(addr(i), src)

6 $new \leftarrow$ `GenSimdExpr`(src)
7 `GenStoreStmt`($addr(i), new,$ '`in loop`')

GenSimdStmt-Epilogue(addr(i), src, ub)

8 $splicePoint \leftarrow (addr(0) + ub * D) \mod V$
9 $new \leftarrow$ `GenSimdExpr`(src)
10 $old \leftarrow$ `vload`($addr(i)$)
11 $spliced \leftarrow$ `vsplice`($new, old, splicePoint$)
12 `GenStoreStmt`($addr(i), spliced,$ '`in epilogue`')

Figure 9: SIMD code generation of statements for the prologue, the steady-state, and the epilogue.

For the prologue, the newly computed values are spliced into the original value prior to the store from byte **ProSplice** to $V - 1$. ProSplice is precisely the alignment associated with the store memory stream, *i.e.*,

$$\text{ProSplice} = addr(i = 0) \bmod V. \tag{8}$$

For the epilogue, the newly computed values are spliced into the original value prior to the store from byte 0 to **EpiSplice**-1, where EpiSplice corresponds to the offset of the first byte after the end of the store memory stream. Since store memory stream is $ub\,D$ bytes long, EpiSplice is computed as,

$$\text{EpiSplice} = (addr(i = 0) + ub\,D) \bmod V. \tag{9}$$

Simdizing a Loop

There is more to simdizing a single statement loop than generating codes for the prologue, steady-state, and epilogue. We must also specify the bounds and the step of the steady-state loop. These steps are detailed below.

- The step of the steady-state loop is set to be the blocking factor B.
- The lower bound of the steady-state loop is set to be the number of original loop iterations being peeled into the prologue, *i.e.*,

$$LB = \lfloor \frac{V - \text{ProSplice}}{D} \rfloor. \tag{10}$$

- The upper bound of the steady-state loop is set to be the original upper bound minus the number of original loop iterations being peeled into the epilogue, *i.e.*,

$$UB = ub - \lfloor \frac{\text{EpiSplice}}{D} \rfloor. \tag{11}$$

4.3 Multiple-Statement Algorithm

Most handling in the SIMD code generation is performed on a per statement basis. Thus, the algorithm in Section 4.2 can naturally handle each statement of a multiple-statement loop. The only exceptions are the loop bound computations in Equations (10) and (11) which clearly need to be computed on a per loop basis. The bounds are thus refined below in the context of multiple-statement loops.

Since Equation (10) computes the lower bound using the alignment of the store, it is not applicable to loops with statements of distinct store alignments. The key observation to address this issue is that we do not need to compute the "precise" lower bound for each statement, as long as each memory operation loads and stores the right data. This is based on our assumption that the loop counter only appears in address computation. Recall that vector memory instructions implicitly truncate the address as they access only aligned data. For example, on AltiVec, loads from addresses `0x1000`, `0x1001`, or `0x100E` each load the same 16 bytes of data starting at `0x1000`.

Exploiting the truncation effect of address computation, we set the lower bound to be the blocking-factor, *i.e.*,

$$LB = B. \tag{12}$$

Equation (12) guarantees that the store associated with each statement in the first iteration of the steady-state loop corresponds to the first *full vector store* of its corresponding stream.

The upper bound specifies the highest iteration in the steady-state loop by which every store in the loop is guaranteed to be a full vector store. For an n-statement loop, we compute the upper bound of the steady-state loop by subtracting the *largest* EpiSplice over all statements from the original upper bound, *i.e.*,

$$UB = ub - \lfloor \frac{\max_{k=1..n} \text{EpiSplice}_k}{D} \rfloor. \tag{13}$$

Furthermore, we need to compute the number of bytes that must be stored in the epilogue, referred to as **EpiLeftOver**. This value is computed on a per statement basis as the total number of bytes in the memory stream, $ub\,D$, minus the number of bytes processed in the prologue, $V - \text{ProSplice}$, and the steady-state loop, $\lceil (UB - LB)/B \rceil\, V$ combined. After simplification using (12), we have

$$\text{EpiLeftOver} = ub\,D + \text{ProSplice} - \lceil \frac{UB}{B} \rceil\, V. \tag{14}$$

For some combinations of ProSplice and ub, EpiLeftOver can be greater than V but is necessarily smaller than $2V$. The epilogue code generation thus has to generate a full vector store followed by a partial one with an epilogue splice point of $(\text{EpiLeftOver} - V)$.

4.4 Runtime Alignments and Upper Bounds

The algorithm that handles VSHIFTSTREAM in Figure 7 generates different code sequences depending on whether a stream is shifted left or right. For runtime alignments, we must introduce VSHIFTSTREAM in such a way that the shift direction can be determined at compile-time in spite of runtime alignments. The zero-shift policy exhibits this property as all misaligned loads are shifted left (to offset 0) and all misaligned stores are shifted right (from offset 0). Therefore we can still use the algorithm in Figure 7 to handle runtime alignment as long as zero-shift policy is applied.

GenSimdStmtSP-Steady(addr(i), src)

```
1  new ← GenSimdExprSP(src)
2  GenStoreStmt(addr(i), new, 'in loop')
```

GenSimdExprSP(n)

```
3  if n ≡ VLOAD(addr(i)) return vload(addr(i))
4  if n ≡ VSPLAT(x) return vsplat(GenExpr(x))
5  if n ≡ VSHIFTSTREAM(src, O_src, to)
6      return GenSimdShiftStreamSP(src, O_src, to)
7  if n ≡ VOP(src_1, ..., src_n)
8      for (k = 1..n) vreg_k ← GenSimdExprSP(src_k)
9      return vop(vreg_1, ..., vreg_n)
```

GenSimdShiftStreamSP(n, from, to)

```
10  shift ← (from − to) mod V
11  if (from > to | Runtime(to))  /* shift left */
12    first ← GenSimdExpr(n)
13    second ← GenSimdExprSP(Substitute(n, i ← i + B))
14  else if (from < to | Runtime(from)) /* shift right */
15    first ← GenSimdExpr(Substitute(n, i ← i − B))
16    second ← GenSimdExprSP(n)
17  GenStoreStmt('old', first, 'in prologue')
18  GenStoreStmt('new', second, 'in loop')
19  GenStoreStmt('old', vload('new'), 'bottom of loop')
20  return vshiftpair(vload('old'), second, shift)
```

Figure 10: Software pipelined SIMD code generation for expressions.

For the lower bound, we can safely use Equation (12) as it solely depends on the blocking factor.

However, we need to refine the upper bound formula (13) as $\max_{k=1..n} \text{EpiSplice}_k$ is expensive to compute at runtime. This can be achieved by finding a suitable upper bound to replace the max term. Recall our assumption that each array is naturally aligned to its data element length. Thus, $addr(i)$ can be represented as $m\,D$ for some integer m. Equation (9) then becomes $(m\,D + ub\,D) \bmod V$ and can be further simplified to $((m + ub) \bmod B)\,D$. According to the definition of mod, the largest value for EpiSplice is thus $(B-1)\,D$.

Replacing the max term in (13) by $(B-1)\,D$, we get this new upper bound:

$$UB = ub - B + 1. \tag{15}$$

Accordingly, (14) can be simplified to

$$\text{EpiLeftOver} = \text{ProSplice} + (ub \bmod B)\,D. \tag{16}$$

Using (16), one can easily prove $\text{EpiLeftOver} < 2V$.

Since the prologue always peels one simdized iteration and the epilogue stores at most 2 full vectors, *i.e.*, two simdized iterations, the simdization is guaranteed to be valid if the original trip count is greater than $3B$. When the trip count is unknown, the simdized codes must be guarded by a test of $ub > 3B$.

4.5 Software Pipelined Algorithm

We can further improve the standard algorithm in Section 4.2 by eliminating the redundant computation introduced during stream shift handling. Recall that, in Figure 9, `GenSIMDShiftStream` combines the values of two consecutive loop iterations, either the current and next iterations for left shifts or the current and previous iterations for right shifts.

For conciseness, we describe here and in Figure 10 the values associated with the smaller iteration count as *first* and the one with the larger iteration count as *second*.

The key idea is to software pipeline the computation of the first and second values. Instead of computing both values associated with the first and second iterations in the loop, we only compute the values associated with the second iteration and preserve them to the next iteration, since this iteration's second values will become next iteration's first values.

As shown in Figure 10, the software pipelined code generation scheme involves the following 3 steps.

1. We precompute $first$ in a non software pipelined fashion (lines 12 and 15) using the standard algorithm `GenSimdExpr`. We then generate a statement to store the values of $first$ to register `old` (line 17), inserted to the prologue of the loop.

2. We compute *second* in a software pipelined fashion (lines 13 and 16). And store *second* to register `new` (line 18). Since this expression is in the loop, we recursively use software pipelined `GenSimdExprSP`.

3. We generate a statement to copy register `new` to register `old` (line 19) at the bottom of the loop.

Note that the steady-state loop involves only the computation of *second* (line 2) and the copy operation between `new` and `old` (line 19). In other words, we have replaced the computation of $first$ in the steady-state loop by a copy operation. Note that the copy operation can be easily removed by unrolling the loop twice and forward propagating the copy operation.

5. EVALUATION

In this section, we present a detailed evaluation of the simdization algorithm. We compare the actual throughput achieved on SIMD units to the bounds derived from code analysis.

5.1 Target Machine Description

The target machine contains generic 16-byte wide SIMD units that are representative of most SIMD architectures currently available. The load-store unit supports 16-byte aligned loads and stores only. Data reorganization is supported by a permute operation that selects arbitrary bytes from two vector registers, similar to the byte permutation operation described in Section 2.2.

5.2 Compiler Infrastructure

Our implementation is based on IBM's xl compiler infrastructure, which supports multiple languages including C, C++, and Fortran, and generates highly optimized codes. The simdization algorithm is implemented in the **Toronto Portable Optimizer** (TPO), the component that performs aggressive dependence analysis and high-level loop transformations. The simdization phase occurs after several loop transformations such as loop interchange and loop distribution that enhance simdization by removing loop-carried dependences along innermost loops. It is followed by other loop transformations that can significantly improve the quality of the simdized codes, notably loop unrolling that removes needless copy operations and a special form of common subexpression elimination, referred to as **Predictive Commoning** (PC) [5], which exploits the reuse among consecutive loop iterations. The back-end code generator (TOBEY) has been extended to target a PowerPC-based processor with SIMD vector units. It performs various target-specific optimizations including instruction scheduling and register allocation.

5.3 Evaluation Methodology

To better evaluate our simdization scheme, experiments were conducted on a set of synthesized loops. These loops heavily stress our heuristics as they exhibit a high ratio of misaligned data references versus computations. The loop benchmarks are synthesized based on a set of parameters, s, the number of statements, l, the number of load references per statement, and, n, the iteration count. Since all arithmetic operations are essentially the same for alignment handling, we use *add* as the sole arithmetic operation in the synthesized loops. The alignment of each memory reference is randomly selected, with a possible bias b ($0 \leq b \leq 1$) toward a single, randomly selected alignment. Each memory reference within a single statement accesses a distinct array, but different statements can contain accesses to the same array. The amount of array reuse r ($0 \leq r \leq 1$, from no reuse to full reuse) among multiple statements is also parameterized.

The metric being used is **operations per datum** (OPD), namely the number of operations needed to compute a single data element. We believe that this metric is more micro-architecture independent than wall time or clock cycles, as it does not depend on the cycle time, instruction latency, and issue width of a particular architecture instantiation. OPD also provides a very intuitive feel for the peek speedup that can be achieved with simdization, e.g. a factor of 4 when 4 integers are packed in a single 16-byte vector register.

We compare the compiler generated code measurement to a **lower bound** (LB) of operations per datum. The lower bound is computed based on parameters $\langle l, s, n, b, r \rangle$. It accounts for the following factors. It includes each *distinct* 16-byte aligned load and store in the loop[3]. The bound also accounts for a minimum number of data reorganizations per statement. This is based on the observation that for a statement with accesses of n distinct alignments, a minimum of $n-1$ `vshiftpair` operations are required. Note that for the shift-zero policy, the number of `vshiftpair` operations is fully deterministic, namely one for each of the m misaligned memory streams. For that policy only, LB reflects m instead of $n-1$. The bound also includes the data computations in the loop, but explicitly ignores all architecture- and compiler-dependent factors such as address computation, constant generation, and loop overhead.

When reporting measurements for the compiler-generated codes, the operations per datum metric includes all overhead present in the execution of the real code, including a single function call and return, address computation, and loop overhead.

5.4 Coverage Analysis

We first evaluate the robustness of our implementation. More than a thousand loops were generated with varying $\langle l, s, n, b, r \rangle$ parameters. In particular, we tested up-to eight

[3]For example, loading `a[i]` and `a[i+1]` anywhere in the loop counts as one when both loads are known at the compile time to map to the same 16-byte aligned memory location

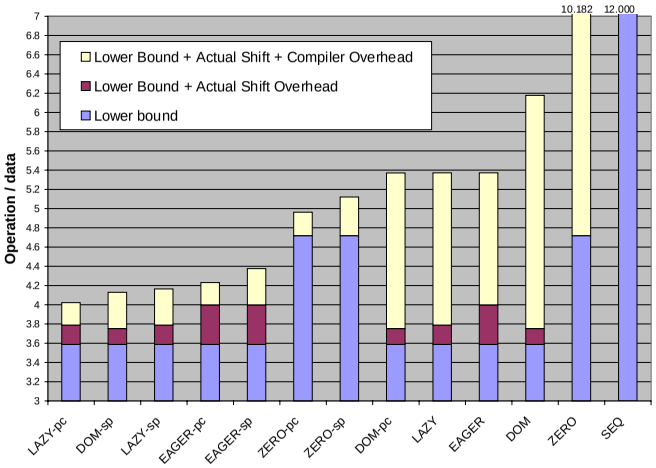

Figure 11: Operations per datum for loops with six loads and one store.

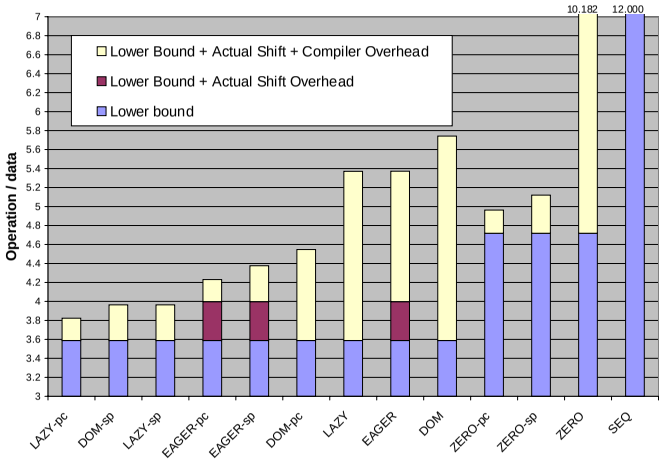

Figure 12: Operations per datum for loops with six loads and one store with OffsetReasoc.

loads per statement, four statements per loop, and a loop trip count in the range of $[997, 1000]$ (for 4-element vectors). The loop count (n), alignment bias (b), the reuse ratio (r) were all randomly selected. Our compiler simdized all the loops. The generated binaries were simulated on a cycle-accurate simulator, and the results were verified.

5.5 Evaluation of Optimization Combinations

This set of experiments evaluates the combinations of shift placement policies and code generation optimizations. Each benchmark used in the experiments consists of 50 distinct loops with identical $\langle l, s, n, b, r\rangle$ characteristics. The results are reported as the harmonic means over all 50 loops. Four shift placement policies are considered, i.e., the zero-shift (**ZERO**), eager-shift (**EAGER**), lazy-shift (**LAZY**), and dominant-shift (**DOM**) policies. The zero-shift policy, although least optimized, is necessary when the alignments are not known at compile-time. Thus measurement of this policy highlights the potential performance degradation due to the lack of compile-time alignment information. Zero-shift policy is also the policy used by prior work [6, 7].

Each shift placement policy can be combined with the following (mostly) orthogonal code generation optimizations.

Software Pipelining (SP) where we directly generate software-pipelined codes to exploit the reuse between consecutive misaligned loads.

Predictive Commoning (PC) where we rely on the more general TPO optimization to exploit the reuse between consecutive misaligned loads.

Memory Normalization (MemNorm) where addresses used in vector memory operations are normalized to their lower 16-byte aligned memory locations to facilitate traditional redundancy elimination optimization.

Common Offset Reassociation (OffsetReassoc) where the associativity and commutativity of the computation are used to group computations with identical offsets to make the lazy-shift and dominant-shift policies more successful.

For conciseness, not all 64 (4 policies times 16 code-gen optimization combinations) results are reported here. Since MemNorm is always beneficial by approximately 0.5% across the board, schemes without it are removed from the figures. Furthermore, using predictive commoning in addition to software pipelining does not bring any additional benefit, thus this combination is also removed from the figures. However, we report data with and without OffsetReassoc separately.

For each data point, we report operation per datum that is broken down into 3 components. The bottom component is the lower bound LB, as defined in Section 5.3. The middle component corresponds to the data reorganization overhead actually introduced by the shift policies over the lower bound. Recall that there is no such overhead for the zero-shift policy because the fully deterministic number of `vshiftpair` operations is fully accounted by LB. The sum of the 3 components is the total cost, as measured by our cycle accurate simulator for the entire program. Thus the top component corresponds to the compiler overhead not accounted for by the previous two components.

We first consider loops with a single integer statement including 6 distinct loads with randomly selected offsets with a bias of 30% (*i.e.*, among the 6 loads, on average 1.8 loads have an identical, randomly pre-selected offset).

Figure 11 shows the resulting operation per datum metric for all significant code generation schemes with OffsetReassoc turned off. When static alignment information is available, our schemes can achieve an opd as low as 4.022 compared to 12 for non-simdized codes[4] (SEQ bar). In general, schemes that introduce redundant operations (i.e. without either PC or SP) perform poorly, with opd ranging from 5.372 to 10.182. Our best schemes also exhibit more than one operation less per datum compared to VAST's approach (ZERO-sp)[7]. Our best schemes perform well with respect to the naive[5] 3.000 bound and the more realistic 3.587 opd bound based on LB.

Further analysis indicates that dominant-shift introduces fewer shifts than lazy-shift or eager-shift, as seen by the contribution of the middle component of each bar. However, the compiler overheads are currently larger for dominant-shift, as seen by the top component of each bar, as it introduces

[4]The original loop has 6 loads, 5 adds, and 1 store.
[5]12 vector operations of 4 integers yields 12/4=3 opd.

Loop Descr.	Align at compile time: Best Policy	Speedup: Actual	Speedup: LB	Align at runtime time: Best Policy	Speedup: Actual	Speedup: LB
S1*L2	LAZY-pc	2.72	3.17	ZERO-pc	2.15	2.36
S1*L4	LAZY-pc	3.02	3.27	ZERO-pc	2.35	2.51
S1*L6	LAZY-pc	3.14	3.35	ZERO-pc	2.42	2.54
S2*L4	DOM-sp	3.42	3.64	ZERO-sp	2.47	2.68
S4*L4	LAZY-sp	3.47	3.64	ZERO-sp	2.43	2.69
S4*L8	DOM-sp	3.71	3.93	ZERO-sp	2.17	2.78

Table 1: Speedup factors of simdized versus scalar codes (4 ints per registers, peek speedup is 4).

Loop Descr.	Align at compile time: Best Policy	Speedup: Actual	Speedup: LB	Align at runtime time: Best Policy	Speedup: Actual	Speedup: LB
S1*L2	LAZY-pc	5.10	5.85	ZERO-pc	4.22	4.63
S1*L4	LAZY-pc	5.49	6.12	ZERO-pc	4.65	4.97
S1*L6	LAZY-pc	5.67	6.25	ZERO-pc	4.83	5.09
S2*L4	DOM-sp	6.06	6.94	ZERO-sp	4.81	5.45
S4*L4	DOM-sp	6.06	6.91	ZERO-sp	4.64	5.43
S4*L8	DOM-sp	6.05	7.32	ZERO-sp	3.88	5.67

Table 2: Speedup factors of simdized versus scalar code (8 short ints per registers, peek speedup is 8).

more redundancy and may generate codes that are more difficult to optimize.

When alignment information is not available at compile time, we must revert to the zero-shift policy, which achieves a 4.963 opd compared to the lower bound of 4.750 opd.

Figure 12 presents the measurements in the same setting as in Figure 11, but with OffsetReassoc turned on[6]. This enables lazy-shift and dominant-shift to have on average no shift overhead over LB, thus resulting in lower overall operations per datum: 3.823, 3.963, and 3.963 for the top 3 schemes compared to 4.022, 4.13, and 4.164 in Figure 11, respectively.

5.6 Efficiency Analysis

This set of experiments investigates the general speedups that can be achieved by our simdization scheme. The measurements are conducted in a wider range of loops, ranging from 1 statement with 2 loads to 4 statements with 8 loads each, all accessing integer arrays. The speedup factors[7] of the simdization are summarized in Table 1. Each row corresponds to the harmonic means over a 50 loop benchmark whose characteristics are summarized in the first column (reuse and bias set to 30%). For each benchmark, we report the speedup factors of the best performing simdization policy for both the compile-time and runtime alignments. For reference, we also indicate an upper-bound speedup factor based on LB.

The general trend in Table 1 is that we achieve higher speedup factors as the loops become more complex, e.g. reaching a speedup factor of 3.71 for a loop with 4 statements with 8 loads each. This is due in part to exploiting more more data locality for the same code size, because each simdized loop iteration covers 4 times more data than original loop iteration and each 16 byte data quantity should be loaded only once using predictive commoning and software pipelining. This is also due in part to the loop overhead becoming smaller for larger loops.

The second general trend in Table 1 is that the lazy-shift policy combined with predictive commoning and the dominant-shift policy with software pipelining are the highest performing policies.

Table 2 indicates the speedup when simdizing loops with short int arithmetic, where 8 shorts are packed in a vector register instead of the 4 ints. Simdization achieves a speedup factor of up to 6.06, compared to a peek speedup of 8 and a more realistic upper bound speedup of 7.32.

[6]Since the loops contains only add operations, our results with OffsetReassoc may be more optimistic than those of real loops where different operations may be involved in an expression which may not be reassociated.

[7]Speedup factor corresponds to the total number of instructions over all loops of the scalar code divided by these of the simdized code.

6. RELATED WORK

There has been a recent spike of interest in compiler techniques to automatically extract SIMD parallelism from programs [4, 6, 8, 9, 10]. This upsurge was driven by the increasing prevalence of SIMD architectures in multimedia processors. Two principal techniques have been used, the traditional loop-based vectorization pioneered for vector supercomputers [11, 12] and the unroll-and-pack approach first proposed by Larsen and Amarasinghe [9]. Our simdization scheme falls into the first category among with others [4, 10, 13, 14].

The most extensive discussion of alignment considerations is in [3]. However, [3] 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 [3, 4]. 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. Even under this condition, our scheme has the advantage of generating simdized prologue and epilogue, which is the by-product of peeling from the simdized loop.

Direct code generation for misaligned references have been discussed by several prior works. [6] described the vectorization of misaligned loads and stores using the VIS instruction set. Their scheme is equivalent to our zero-shift placement policy plus the non software pipelined code generation algorithm. It is not clear whether multiple statements, runtime alignments, and unknown loop bounds can be handled. In [4], Bik *et al.* described a specific code sequence of aligned loads and shuffle to load memory references that cross cache line boundaries, which is implemented in Intel's compiler for SSE2. However, their method is not discussed in the context of general misalignment handling. Furthermore, neither [6] nor [4] exploit the reuse when aligning a stream of contiguous memory. As shown in our evaluation in Figure 11, without exploiting the reuse, there can be a performance slowdown of more than a factor of 2.

The most extensive alignment handling was implemented in the VAST compiler [7]. Like our scheme, VAST is able to simdize loops with multiple misaligned references, unknown loop bounds, and runtime alignments, and exploit the reuse when aligning a steam of contiguous memory. However, there is no public information available regarding their

alignment handling scheme. We can only conjecture, from the simdized codes produced by the compiler, that VAST's scheme is equivalent to our zero-shift policy combined with software pipelining. Therefore, our scheme has the advantage of the other three additional placement policies.

An interesting simdization scheme using indirect register accesses is discussed in [10]. However, their method is specific to the eLite processor that supports gather and scatter within a special register file, which is not applicable to typical MME processors. In [15], register packing and shifting instructions were used to exploit temporal and spatial reuse in vector registers. Their definition of *replicate* and *shift-and-load* is very similar to our `vsplat` and `vshiftpair` operations. However, their work does not address alignment handling.

In earlier work on compiling for distributed memory systems, [16] describes the Alignment-Distribution Graph that has some features in common with our data reorganization graph, notably the existence of nodes to specify transformation of data related to alignment or placement in memory.

7. CONCLUSION

This paper proposes a compilation scheme to vectorize misaligned references for SIMD architectures that support only aligned loads and stores. In our framework, a loop is first simdized as if the memory unit imposes no alignment constraints. The compiler then inserts data reorganization operations to satisfy the actual alignment requirement of the hardware. Finally, the code generation algorithm generates SIMD codes based on the data reorganization graph, addressing realistic issues such as runtime alignments, unknown loop bounds, residue iteration counts, and multiple statements with arbitrary alignment combinations.

Beyond generating a valid simdization, we investigate methods to further improve the quality of the generated codes. We propose four stream-shift placement policies to minimize the number of data reorganization generated by the alignment handling. And our code generation algorithm exploits the reuse when aligning a stream of contiguous memory using software pipelining techniques.

We demonstrate near peak speedup for a set of loops where 75% of the static memory references in the loops are misaligned. Comparing the dynamic instruction count of simdized codes to an idealistic scalar instruction count, we achieve speedup factors of up to 3.71 and 6.06 for vectors packed with 4 and 8 data, respectively.

Many further issues need to be investigated to match the wide range of situations that are only handled at present by experienced assembly programmers. Examples of such issues are alignment handling of loops with non-unit stride accesses, accesses to scalar variables including induction variables occurring in non-address computation, non-naturally aligned arrays, and data conversions that require packing and unpacking capability. While these issues were not addressed by this paper, we believe that our approach based on data reorganization graphs provides a solid foundation to solving these problems in the future.

8. REFERENCES

[1] Gang Ren, Peng Wu, and David 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.

[2] Motorola Corporation. AltiVec Technology Programming Interface Manual, June 1999.

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

[4] 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.

[5] Kevin O'Brien. Predictive Commoning: A Method of Optimizing Loops Containing References to Consecutive Array Elements. In *IBM Interdivisional Technical Liason*, 1990.

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

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Thomas J. Sheffler. Modeling Data-Parallel Programs with the Alignment-Distribution Graph. *Journal of Programming Languages*, 2(3):227–258, 1994.