H-0236 (H0512-002) November 30, 2005
Computer Science

IBM Research Report

Multi-Platform Auto-Vectorization

Dorit Naishlos, Richard Henderson*
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

*Red Hat

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

Multi-Platform Auto-Vectorization

Dorit Naishlos
IBM Haifa Labs
dorit@il.ibm.com

Richard Henderson
Red Hat
rth@redhat.com

Abstract. The recent proliferation of the Single Instruction Multiple Data (SIMD) model has lead to a wide variety of implementations. These have been incorporated into many platforms, from gaming machines and embedded DSPs to general purpose architectures. In this paper we present an automatic vectorizer as implemented in GCC - the most multi-targetable compiler available today. We discuss the considerations that are involved in developing a multi-platform vectorization technology, and demonstrate how our vectorization scheme is suited to a variety of SIMD architectures. Experiments on four different SIMD platforms demonstrate that our automatic vectorization scheme is able to efficiently support individual platforms, achieving significant speedups on key kernels.

1 Introduction

In recent years a wide range of modern computational platforms have incorporated Single Instruction Multiple Data (SIMD) extensions into their processors. SIMD capabilities are now common in modern digital signal processors (DSPs), gaming machines, and general purpose processors. These SIMD mechanisms allow architectures to exploit the natural parallelism present in applications by simultaneously executing the same instruction on multiple data elements that are packed in "vector" registers. Optimizing compilers use vectorization techniques [20] to exploit the SIMD capabilities of these architectures. Such techniques reveal data parallelism in the scalar source code and transform groups of scalar instructions into vector instructions. However, it is often complicated to apply vectorization techniques to SIMD architectures [15]. This is because SIMD architectures are largely non-uniform, supporting specialized functionalities and a limited set of data types. Vectorization is often further impeded by the SIMD memory architecture, which typically provides access to contiguous memory items only, with additional alignment restrictions. Overcoming these architectural limitations requires special mechanisms that widely differ from one SIMD platform to another. A vectorizing compiler must be aware of these capabilities with their associated costs and use them appropriately to produce high quality code.

In this paper we discuss the challenges and considerations that are involved in developing an auto-vectorizer that is high-level, generic and multi-platform, and on the other hand, makes low-level platform-dependent decisions to generate the most efficient SIMD code for each target platform. We present some of the high-level abstractions we introduced into the GCC intermediate language (IL) in order to express some of the unique architectural features of SIMD platforms, and demonstrate how these abstractions meet the requirements of several SIMD architectures. To the best of our knowledge, this is the first effort that applies vectorization techniques in a truly multi-platform setting which involves several different SIMD extensions. Such an effort facilitates a comprehensive comparison of the key architectural features of various SIMD platforms, presented here together with actual code examples and experimental results that are generated for several different SIMD architectures using one vectorizing compiler. Furthermore our compilation platform is freely available, allowing anyone to reproduce the techniques presented in this paper on any SIMD platform supported by GCC.

The paper is organized as follows: Section 2 provides background on our compiler and vectorizer and discusses the multi-platform aspects of our work. Sections 3 and 4 describe how the challenges and considerations that were introduced in Section 2 are put in practice to support two specific features: alignment and reduction. In Section 5 we present experimental results, demonstrating significant performance improvements across 4 different platforms (Figure 7). Section 6 presents related work, and section 7 concludes.

2 GCC and the Multi-Platform Vectorization Challenge

2.1 GCC Overview

The GNU Compiler Collection (GCC) project of the Free Software Foundation [6] is one of the most widespread and multi-targetable compilers in use today, and is the de-facto standard in the Linux eco-system. GCC uses multiple levels of intermediate languages (IL) in the course of translating the original source language to assembly. Traditionally, GCC performed its optimizations on a single IL called the Register Transfer Language (RTL) which is fairly low-level, and not amenable to complex transformations. During the GCC 4.0 development cycle, a new higher level intermediate language called GIMPLE was introduced [11]. This new IL supports Static Single Assignment (SSA) [14] and retains enough information from the source language to facilitate advanced data-dependence analysis and aggressive high-level optimizations. The high-level target-independent GIMPLE statements are later translated to RTL instructions which directly correspond to the target instruction set (see Figure 1). This is where target-dependent optimizations are applied, such as instruction scheduling, software pipelining and

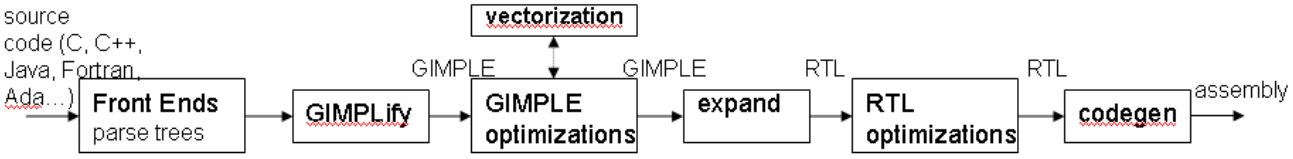

Figure 1. High-level structure of GCC

register allocation.

Each platform that GCC supports has a target-specific "port" that models the target processor instructions using `machine description` (md) files. When building the compiler, a set of data structures is initialized according to the information in these files. One such data structure is the `optabs` (operation tables), where GCC records target-specific information for a set of generic operations. The optabs provide a means for the compiler optimizations to query which operations are supported on the target platform, and for which operand types. As the following sections reveal, in order to express some vector concepts in GCC we added new operations (optabs) to GCC, and new IL idioms (operation-codes) to represent these operations in GIMPLE. Using this mechanism GCC can be configured to generate code for more than thirty different computer architectures. This flexibility and portability is one of the reasons that use of GCC has become pervasive throughout the software industry.

2.2 The GCC Vectorizer

On January 1st, 2004, we provided the first version of auto-vectorization in GCC [5], which is now part of the official GCC 4.0 release. The auto-vectorizer currently applies to countable innermost loops of arbitrary counts that may contain some control-flow constructs (which are if-converted), and that contain accesses to consecutive memory addresses which may or may not be aligned. Additional capabilities are constantly being developed. Among the most recent additions introduced to GCC 4.1 are the auto-vectorization of reduction operations and the use of loop-versioning as another means of handling misaligned accesses.

One platform dependent parameter that is important for vectorization is the vector size (VS), which each relevant platform needs to define (in its md file). The number of elements that can fit in a vector determines the Vectorization Factor (VF), the amount of data parallelism that can be exploited. A vector of size 16 bytes for example, can accomodate 16 byte-elements (chars), 8 half-word-elements (shorts), and 4 word-elements (ints). Table 1 summarizes this information for a set of GCC targets, along with the data-types that each target supports

Platform	ISA extension	VS	Data Types
IBM/Motorola/Apple PowerPC	AltiVec	16	char, short, int, float
Intel/AMD ia32	SSE	16	char, short, int, float, double
Intel/AMD ia32	MMX, 3dNow!	8	char, short, int, float
Intel ia64		8	char, short, int, float
Freescale E500	SPE	8	char, short, int, float
MIPS	MDMX, MIPS3D	8	char, short, int, float
Sparc	Sparc VIS	8	char, short, int
HP Alpha	MAX	8	char

Table 1. SIMD platforms summary

in its SIMD unit. Alpha for example does not have a SIMD unit, but can support a few operations (max, min and widening) on multiple chars in parallel. Some operations can be vectorized without a SIMD unit, by exploiting sub-word parallelism in data types like int (e.g. operate in parallel on 4 chars packed in one int). For example, it is possible to perform addition for chars on the Alpha , with the required bit manipulations done in software.

2.3 Introducing vector operations to the GIMPLE IL

In GIMPLE, vector operations are generally represented like scalar operations — the same operation-codes are used, but the arguments are of vector type. This is suitable for "pure SIMD" operations, in which the functionality represented by the operation-code (e.g. addition) is performed on each element of the vector. Other vector operations like reductions, alignment-support mechanisms and vector element shuffling operations, are meaningless in the context of scalar computations and therefore were not available in GIMPLE. In order to express these mechanisms in the vectorized GIMPLE IL, we had to introduce some new platform-independent abstractions. While the high-level representation in GIMPLE simplifies analyses critical for vectorization such as alignment, aliasing and loop-level data-dependences, GIMPLE is less appropriate for accessing target specific information, and expressing target-dependent constructs. This section discusses the issues and considerations involved in introducing new vector operations to GCC and the GIMPLE IL.

Generality vs. applicability: When introducing a new operation, a natural desire is to use an abstraction as general as possible. This approach minimizes increase of operation-codes by allowing to reuse them for different purposes. For example, a general data-permutation abstraction can be used to represent alignment handling, data packing/unpacking for type conversions, table lookups, and more). A general abstraction may also be useful for representing powerful mechanisms that future targets may need to express. On the other hand, a general abstraction may be too general for certain targets. Indeed, an interesting challenge arises when one platform supports mechanisms that are too general compared to what is available to other platforms. Going back to the

permutation example, most SIMD targets support only a special case of permutations, but not a general arbitrary permute such as available in Altivec. In such cases we need to bridge the gap between the different platforms. In the `perumte` case, we avoid introducing a generic arbitrary permutation for expressing functionality that can be addressed with simpler mechanisms that better match the technology available to the target platforms (e.g. the `realign_load` in Section 3) Finally, one should also consider what kind of operations an automatic vectorizer will be able to exploit. In cases where the vectorizer can use only a limited functionality, it is less useful to introduce a more generic form.

Compound operations vs. basic operations: SIMD architectures often have specialized support for certain compound operations , such as vector subtract and saturate. In principle, it is desirable to avoid introducing unnecessarily complicated operations to the IL if they could be represented with a sequence of more basic operations. Using basic operations not only reduces the number of new operation-codes, but also reduces the number of ways a certain functionality can be represented, thus making it easier for the optimizers to determine the canonical representation of expressions in the IL. In addition, exposing the basic operations can increase the effectiveness of optimizations as opposed to having one "black-box" operation. On the other hand, it is often the case that if the higher-level idiom is detected and conveyed in the IL, better code can be generated, as some targets often directly support the compound operation but do not directly support the more basic functionalities that it encapsulates. This is the case for example in reduction idioms like dot-product and sum-of-absolute-differences, that are often directly supported by targets, but the basic operations that these idioms encapsulate, namely widening-summation and widening-multiplication, have less efficient support.

IL conventions: Attention must also be paid to existing GCC operations and conventions. Specifically, we want to (1) reuse existing optabs and operation-code where applicable, (2) use conventional semantics and default values (for example, shift operations in GCC specify the shift amount in bits; while it could be useful to introduce a vector byte-shift operation, as in section 4, this is inconsistent with the rest of the compiler and may be confusing), (3) use terminology that does not conflict with existing terms of SIMD extensions on different targets or existing RTL operation-codes, and (4) provide clear definitions of the new operations and avoid ambiguity (e.g. with respect to endianess).

Performance: Last but not least, we always strive to generate statements that will eventually translate into the most efficient instructions available on a target.

3 Alignment Idioms

One of the difficulties that arises when vectorizing for SIMD is the alignment constraints of the SIMD memory architecture. Accessing a block of memory from a location which is not aligned on a natural vector size boundary is almost always prohibited or bears a heavy performance penalty. These memory alignment constraints can be handled in software using data reordering mechanisms. Such mechanisms are costly, and usually involve generating extra memory accesses and special code for combining data elements from different vectors. The vectorizer must be aware of the available alignment mechanisms in order to determine whether vectorization is possible and profitable. It also needs to generate code that correctly and efficiently accesses data located at potentially unaligned memory addresses, which implies that low-level alignment handling constructs need to be abstracted and expressed in the GIMPLE IL.

In devising our alignment handling scheme we tried to identify common alignment mechanisms between targets and abstract them into alignment handling schemes that are explicitely expressed in GIMPLE. As demonstrated below, SIMD alignment mechanisms vary greatly from platform to platform, but there is also a lot of commonality between groups of platforms. For clarity, we focus in this section on unaligned loads; the case of unaligned stores is largely symmetric.

3.1 Overview of alignment mechanisms across different platforms

Alignment-related mechanisms in SIMD platforms support one or more of the following capabilities: (1) memory read/write to an unaligned memory location, (2) general vector shuffling utilities (not specific to alignment handling), and (3) specialized alignment support. This section provides an overview of the available capabilities in these three categories across different SIMD platforms. We refer to Figure 2 to illustrates some of these capabilities. The figure shows an access to an unaligned address (assuming vector size VS=16 bytes, and 4 byte elements, laid out in memory starting from aligned address 0). Trying to fetch the data elements [c,d,e,f] into a vector register results in an unaligned load from address 8.

(1) Unaligned memory accesses. The behavior of a vector load or store instruction given an unaligned address differs from platform to platform. The Intel MMX and SSE ISA extensions are the only SIMD platform that have instructions that will properly operate with an unaligned address and will directly fetch elements [c,d,e,f]. However, these instructions are more costly than aligned memory accesses and should be avoided whenever possible. Most SIMD platforms have load and store instructions that drop the low order bits of the address, accessing VS

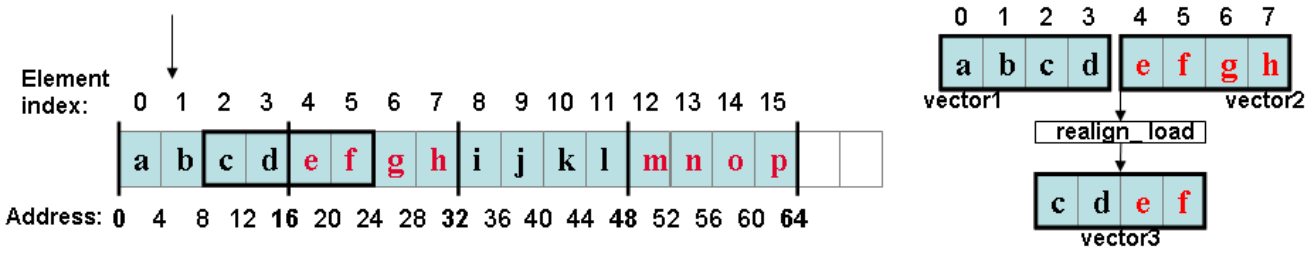

Figure 2. Unaligned access. Left: data in memory. Right: data in vector registers

bytes starting from the "floor aligned" address (the closest previous aligned address). A load from address 8 in Figure 2 using these instructions will fetch elements [a,b,c,d]. This behavior can be seen in the AltiVec, MIPS-3D, and Alpha instruction sets. Lastly, some platforms will simply trap upon access to an unaligned address; examples here are IA-64 and SPE.

(2) General vector-merge mechanisms. A logical way to handle misaligned accesses makes use of general utilities to shuffle elements between vectors, using mechanisms like vector shifts, rotates or permutes. The general idea is to access the neighboring aligned locations just before and after the address in question, load the two sets of elements into vectors, and extract and merge the relevant elements from there. Looking at the example in Figure 2, this means loading elements [a,b,c,d] from the aligned address 0 into one vector, loading elements [e,f,g,h] from the aligned address 16 into another vector, and extracting elements [c,d] and [e,f] from these two vectors and merging them into a third vector.

This can be done using a double-vector shift operation. It takes two vectors and a shift amount as parameters, and treats the data in the two input vectors as one stream of elements (as if they are concatenated) and shifts this stream as a whole by the shift amount. Using the misalignment value as the shift amount (8 bytes in the example) would bring the desired elements into one vector. Unfortunately, while most SIMD targets have single-vector shifts (shiting bits within a single vector), few have double-vector shifts. Even then, the shift amount may be restricted to immediate shift counts, and therefore cannot be used if the misalignment amount is unknown at compile time.

Another general mechanism that can be used is a `permute` operation. As with a double-vector shift operation it also extracts one vector from two vectors. An additional argument is a permutation mask that, for each of the elements in the output vector, specifies which element from the two input vectors to copy over to each element of the output vector. In our example in Figure 2, a `permute` operation with mask (2,3,4,5) will extract the desired elements at these indices in the concatenated vector. Permutes can also be used to implement vector shifts, and for

other purposes beyond handling alignment.

The most important thing to notice about the shift and permute idioms is that on most platforms these operation are not supported in their generic form, and usually require immediate values to the instructions, or allow only a fixed permutation. Only the AltiVec `vperm` instruction allows arbitrary, variable permutation. Other platforms (MMX, SSE, MIPS, IA-64 and SPE) either have instructions to merge the high and low parts of two vectors, or something akin to the AltiVec `vperm`, but accepts only an immediate constant in the instruction.

(3) Specialized realignment mechanisms. Some targets offer additional specialized instructions to handle unaligned accesses. The MIPS MDMX integral vector extension is unique in using load-left and load-right instructions to handle misaligned memory operations. In all other cases the platforms use data from two sequential floor-aligned addresses, some function of the misaligned address, and some instruction sequence to merge and extract the desired results.

The simplest form of this is found in MIPS64 in the `alnv.ps` instruction, where the function of the misaligned address is 8 times the bottom three bits, and this one instruction is the whole of the sequence. In this case, the instruction extracts the low 3 bits from a register holding an address and uses that as input to a shifter operating on two registers holding the input vectors. A slightly more complicated form exists on Alpha, where a sequence the `extql` and `extqh` instructions may be used together with an `or` instruction to perform the same operation as the MIPS `alnv.ps`. The most general form of the "function of the misaligned address" exists in the AltiVec's `lvsr` instruction. In this case the instruction extracts the bottom four bits and computes a vector result appropriate for input as a permutation mask to the `vperm` instruction discussed previously.

3.2 Abstract GIMPLE alignment handling scheme

Amid the diversity of the alignment mechanisms presented above, it is possible to identify classes of mechamisms that can be mapped into two general alignment handling schemes, along with appropriate GIMPLE abstractions. The key factor that differentiates the two mechanisms is whether or not they allow exploitation of data reuse across loop iterations. As described above, some platforms handle alignment by loading two consecutive vectors such that each step of the computation loads a vector that was also loaded by the previous iteration. Going back to Figure 2, we load data from addresses 0 and 16 in order to fetch elements [c,d,e,f]; we then load data from addresses 16 and 32 in order to fetch [g,h,i,j], thus loading from address 16 twice. If we can reuse the this data, we can limit the computation within each step to a single vector load and a merge operation, as shown in Figure 3b.

Target	misaligned-load	align-ref load	realign	rt
AltiVec		lvx	vperm	lvsr
SSE	movdqu			
MIPS-3D		luxc1	alnv.ps	address
MIPS64	ldl, ldr			
alpha		ldq_u	extql, extqh, or	address

Table 2. Unaligned load support across platforms

When a platform has alignment mechanisms that allow this kind of optimization, we try to expose the relevant buiding blocks of the scheme, so that they can be moved around by the GIMPLE optimizers. For that, we need to introduce GIMPLE idioms that abstract these building blocks. We refer to this alignment handling scheme as the "software-pipelined realignment" scheme, as it has the effect of software-pipelining the vector loads. When the alignment mechanisms that a platform provides have to be executed as one batch, there is no advantage in exposing them to GIMPLE, and we use a single idiom to abstract them. We refer to this scheme as the "direct realignment" scheme.

3.2.1 The direct realignment scheme

This scheme uses a single idiom to abstract the actual implementation of the misaligned access. For that we needed to introduce a data-reference idiom that will represent an unaligned access, and that we could distinguish from a regular data-reference to an aligned address.

Loads and stores are represented in GCC as a `move` operation, that moves data between memory (represented by a data-reference operation-code. e.g. `array_ref`) and a register. The vector memory-references we create during vectorization are pointer based references, represented by the (existing) operation-code `indirect_ref(ptr)`. We use it when the memory-reference is aligned, and a new operation-code - `misaligned_indirect_ref(ptr,mis)` to express an access to an address (pointed to by `ptr`) whose misalignment is `mis`. `mis` is an integer constant; when the misalignment is unknown at compile time, we set it to 0. We also introduced a new kind of move operation to GCC - `movmisalign`, which also serves as an API for the vectorizer to query whether this kind of functionality is supported by the target. The subsequent RTL expansion routines expand the `misaligned_indirect_ref` idiom to the appropriate code sequence for each plaform. Column "misaligned-load" in Table 2 shows the code that a `misaligned_indirect_ref` corresponds to on each platform (if available).

(a) realignment idioms, unoptimized:

```
LOOP:
vec1= align_ref(addr_i)
vec2= align_ref(addr_i+VS-1)
vec3= realign(vec1,vec2,rt)
addr_i += VS
END_LOOP
```

(b) optimized, with a target hook:

```
vec1= align_ref(addr_0)
rt= call gen_RT(addr_0)
addr_i= addr_0+VS-1
LOOP:
vec2= align_ref(addr_i)
vec3= realign(vec1,vec2,rt)
vec2= vec1
addr_i += VS
END_LOOP
```

(c) optimized, w/o a target hook:

```
vec1= align_ref(addr_0)
addr_i= addr_0+VS-1
LOOP:
vec2= align_ref(addr_i)
vec3= realign(vec1,vec2,addr_i)
vec2= vec1
addr_i += VS
END_LOOP
```

Figure 3. Software-pipelined realignment

3.2.2 The software-pipelined realignment scheme

Two main building blocks are required for the software-pipelined realignment. The first is a mechanism to generate floor aligned memory references. For this we create the operation code `align_indirect_ref`. Column "align-ref load" in Table 2 shows how this idiom is supported on different platforms.

The second building block is a mechanism to express a merge-like operation of two vectors, extracting the relevant data elements accoding to the misalignment of the address. For this we create the operation code `realign_load`; it takes three arguments: two vectors, and a `realignment_token`. The `realignment_token` can be an address, a bit mask, a vector of indices, an offset, or anything that can be generated as a function of the respective address. We define the functionality of `realign_load` in terms of `mis` - the misalignment of the address (i.e. address&(VS)), as follows: The last `VS-mis` bytes of vector vec1. are concatenated to the first `mis` bytes of the vector vec2. Figure 3a illustrates how these idioms are to be used in order to support unaligned accesses.

Consider the pseudo-code in Figure 3. Note that the offset for the second load is $\mathtt{VS} - 1$, not VS. This is done to avoid reading beyond the boundaries of the array in the case that the end of the array is page aligned. Using $\mathtt{VS} - 1$ as an offset is just a more efficient way of advancing the address by VS guarded by an test that checks that the address is unaligned.

The form of the realignment token (RT) can be very different between systems. As shown in Table 2, for Altivec RT is a permutation mask expressed using a vector, and for Alpha and MIPS is is an address. We therefore wanted to avoid predefining a type and semantics for RT. We accomplish this by introducing a target intrinsic to determine the type of RT. The details are hidden from the vectorizer and GIMPLE behind a function call (the call to `gen_RT` in Figure 3b). This function call is later expanded to the appropraite RTL sequence for each target.

The key element about the `realign_load`, that lets it be general enough and yet not too generals, hide low-level details while maintaining low register pressure and allowing each target to express the best alignment handling capabilities it has, is the abstraction that the `realignment_token` provies.

We also provide a default interpretation of the `RT` if a target can support the `realign_load` idiom but doesn't have a specialized way to generate the `RT`. In this case we use the address as the `RT`, as shown in Figure 3c. As illustrated in Table 2, targets like SSE that require `RT` to be a compile-time constant cannot use the software-pipelined scheme when the alignment is unknown.

3.2.3 Other alternatives

The alignment handling scheme described above was designed after careful consideration of several alternatives. We briefly describe them here.

Hiding the details, option 1: As illustrated in Figure 3a, we could have kept the sequence that constitutes the software-pipelined realignment support intact, hiding it entirely behind the `misaligned_ref` idiom, relying on subsequent RTL level passes to find the data reuse across iterations (after the GIMPLE `misaligned_ref` is expanded into a sequence of RTL instructions) using optimizations like predictive commoning, and superword-register caching [17]). However, the most appropriate place for such otimizations in GCC is GIMPLE, where most of the loop analysis engine is implemented. Therefore, the idioms we presented would have to be exposed in GIMPLE and explicitly generated by the vectorizer. In doing so, the vectorizer can easily generate the optimized code rather than leave it for post-vectorization passes to rediscover the optimization potential, especially as we want the vectorizer to be aware of the costs involved in the transformation it performs.

Hiding the details, option 2: Another alternative that also hides the alignment handling details, is to provide target-specific hooks in the vectorizer that let each port generate the most efficient code for its target, at the GIMPLE level, using black-box target specific functions to express alignment mechanisms. This scheme is disadvantageous in several respects. First, a lot of the functionality that is common to many targets would have to be duplicated on multiple target ports. Second, the vectorizer, as well as the rest of the GIMPLE optimizations, would be unaware of the semantics of these black-box functions, and would have difficulty estimating the overall cost of applying vectorization, and optimizing across these function calls. We tried to minimize the use of such target specific functions, and introduced only the `gen_RT`.

Considering register pressure: Several different alternatives could be used to set the third argument of a `realign_load`. The ones chosen are those that don't unnecessarily increase the register usage in the loop. This is why (as shown in Figure 3c) we use the address accessed at the same iteration as the `realign_load`, (`addr_i`) rather than the initial address (`addr_0`) or the misalignment, which will cause following passes to use two registers for this loop when only one is needed. This is also why we let targets define a specialized RT. Indeed, the RT is calculated as a function of the address, and potentially an address could always be used as the third argument to `realign_load`, hiding the intermediate step of generating the RT from GIMPLE. However this scheme would end up either increasing register pressure if `addr_0` is used, or hindering loop invariant code motion (i.e. the RT generation code) if instead `addr_i` is used.

Using a more generic permute: The functionality accomplished by the `realign_load` idiom can be represented using a more generic permute-like operation, which could be utilitized for other purposes beyond alignment handling. In other words, we could define an idiom that can extract any permutation of elements, and not necessarily a consecutive permutation of elements. However this solution is not suitable for GCC, because a permute idiom has to take a permutation mask as an input, which means we would have to expose what RT means, whereas many targets do not use a permutation mask as an input to their realignment mechanism. Also recall that most targets don't implement a generic permute, but provide only specialized cases of permute. Therefore, realignment, as well as other potential uses of a permute, would be more appropriately addressed in GCC via specialized idioms rather than a general permute.

3.3 Putting it all together

Vectorization of misaligned accesses using the idioms described above is only the last step of alignment handling. Due to the penalties associated with these mechanisms, techniques like loop peeling and static and dynamic alignment detection [1, 9, 7] are often used to eliminate misaligned accesses. Alignment handling therefore consists of three steps: (1) static alignment analysis. (2) transformations to force alignment; The GCC vectorizer uses loop versioning with run-time alignment checks, and loop peeling, in order to force the alignment of data references in the loop. Note that peeling can be used to force the alignment of only a single data reference (DR), so the vectorizer needs to choose which DR to peel for. (3) if data-references which are not known to be aligned still remain after peeling/versioning, the vectorizer will proceed to vectorize the loop only if the target platform provides mechanisms to support misaligned accesses, using the realignment idioms.

(a) Altivec: software-pipelined realignment:

```
scalar_peel_loop:
 mis = min(16-p&15,0);
 for(i=0;i<mis;i++){
  p[i] = p[i] + q[i];
 }
alignment_setup:
 neg r9,r9
 lvsr v11,r0,r9
 lvx v12,r4,r7
 add r10,r4,r0
vectorized_loop:
 li r9,0
 for(i=mis;i<n/16;i++){
  lvx v0,r10,r9
  lvx v1,r9,r11
  vperm v13,v12,v0,v11
  vor v12,v0,v0
  vaddubm v1,v1,v13
  stvx v1,r9,r11
  addi r9,r9,16
  bdnz
 }
```

(b) SSE: direct realignment:

```
scalar_peel_loop:
 mis = min(16-p&15,0);
 for(i=0;i<mis;i++){
  p[i] = p[i] + q[i];
 }
vectorized_loop:
 for(i=mis;i<n/16;i++){
  movdqu (%edi,%edx),%xmm0
  paddb (%edx,%eax),%xmm0
  incl %ecx
  movdqa %xmm0,(%edx,%eax)
  addl $16,%edx
 cmpl %esi,%ecx
  bj
 }
```

(c) alpha: direct realignment:

```
scalar_peel_loop:
 mis = min(8-p&7,0);
 for(i=0;i<mis;i++){
  p[i] = p[i] + q[i];
 }
vectorized_loop:
 for(i=mis;i<n/8;i++){
 addl r31,r6,r1
 addq r24,r1,r5
  ldq r4,0(r5)
 addq r1,r23,r1
  ldq_u r3,0(r1)
  ldq_u r2,7(r1)
  extql r3,r1,r3
  extqh r2,r1,r2
  bis r3,r2,r3
  and r4,r22,r1
  and r3,r22,r2
  addq r1,r2,r1
  xor r4,r3,r4
  and r4,r0,r4
  xor r1,r4,r1
  stq r1,0(r5)
  addl r6,8,r6
 }
```

(d) ia64: no realignment - use loop versioning:

```
if (p and q are aligned) {
vectorized_loop:
 for(i=0;i<n/8;i++){
  ld8 r16 = [r14], 8
  ld8 r14 = [r15] ;;
  padd1 r14 = r14, r16 ;;
  st8 [r15] = r14, 8
 }
}else {
scalar_loop:
 for(i=0;i<n;i++){
  p[i] = p[i] + q[i];
 }
}
```

Figure 4. Alignment handling code generated afor different platforms

Figure 4 illustrates the different alignment handling schemes generated for different targets. As an example we use a simple char-addition loop: $for(i = 0; i < n; i++)p[i] = p[i] + q[i]$. `p` and `q` are pointers passed as arguments to the function. For Altivec, SSE and Alpha, the read and write to `p[i]` are aligned using loop peeling. The remaining (potentially) unaligned read from `q[i]` is handled using the mechanisms available to each target. For ia64 we resort to loop versioning with a run-time alignment check to guard which loop will be executed. For simplicity we use a constant loop bound, replace some of the assembly with high-level pseudo-code, and add labels.

4 Reduction idioms

Reduction is a computation on the elements of a vector that obtains a single scalar result. Summing the elements of a vector, or finding the maximum/minimum element in a vector are examples for reduction operations. Reduction computations involve a cross-iteration dependency, but in some cases (if the operation is commutative and associative, and the intermediate values of the computation are not used), the reduction can be vectorized by

changing the computation order and accumulating partial results, that are "reduced" into the final result after the loop.

Vectorizing a reduction consists of initialization before the loop, partial-results computation inside the loop, and a reduction finalization at the loop epilog. The following subsections discuss the issues that arise when trying to define generic idioms to represent these three parts of the computation.

4.1 Reduction initialization

Consider the cases of a summation reduction and a product reduction: $sum = x; for(i...)sum+ = a[i]$ and $prod = y; for(i...)prod* = a[i]$. There are two alternatives for initializing the partial-results vectors for these reduction comptations: (1) Initialize the vectors to [x,0,0,0] ([y,1,1,1]) for the summation (product). (2) Initialize the vectors to [0,0,0,0] ([1,1,1,1]) for the summation (product), and adjust the final scalar result at the loop epilog, by adding x (multiplying by y). The second option is often more efficient as targets usually have a much easier way to initialize a vector with the same constant value than anything else. (e.g. Altivec's `vsplti*` instruction). At present this is the scheme that the vectorizer is using.

4.2 Partial sums

The initialized vector described above feeds the vector operation in the first iteration of the loop. This vector operation accumulates the partial results in the loop, and is represented trivially by using the existing GIMPLE operation-codes (addition, multiplication, maximum, minimum, etc.) operating on vector-type elements, The following step needs to operate across the elements of the last vector of partial results to produce the final scalar value and is described next.

4.3 Reduction epilog

There are three different ways to implement the reduction epilog: (1) use a specialized `reduc_op` idiom that is directly supported by the target, (2) compute the results using vector shifts, or (3) compute the result sequentially.

The vectorizer can choose which epilog scheme to generate according to available target support. Alternatively, it can defer the decision until RTL expansion, and always produce a black-box `reduc_op` during vectorization (ignoring which of the three schemes will be expanded). Since the epilog can be quite costly, we decided to expose these alternatives to the vectorizer, to allow it to properly evaluate the costs that are involved in vectorizing the

(a) using reduc_op:

```
va_3=phi(va_2)
va_4=reduc_op(va_3)
a_5=bitfield_ref(va_4,0)
...=use(a_5)
```

(b)using shifts:

```
va_3=phi(va_2)
va_4=vec_shift(va_3,64)
va_5=vop(va_3,va_4)
va_6=vec_shift(va_5,32)
va_7=vop(va_5,va_6)
a_5=bitfield_ref(va_7,bitpos)
...=use(a_5)
```

(c) using scalar operations:

```
va_3=phi(va_2)
a_6=bitfield_ref(va_7,0)
a_7=bitfield_ref(va_7,32)
a_8=bitfield_ref(va_7,64)
a_9=bitfield_ref(va_7,98)
a_5=op(a_6,a_7,a_8,a_9)
...=use(a_5)
```

Figure 5. Three possible reduction epilogs (for vector of 4 words)

loop. Also this can potentially allow GIMPLE optimizations to optimize this code. The three alternatives for implementing the reduction epilog are illustrated in Figure 5, and discussed below.

`reduc_op` epilog. The `reduc_op` in Figure 5a stands for any specialized reduction idiom that is directly modeled in the target's md file. For example, a commonly supported operation among SIMD architectures is the sum across the elements of a vector, that can be directly used to reduce the partial results into the final sum. We repserent this operation with a `reduc_plus` operation-code.

vector-shifts epilog Targets that support whole vector shifts, can implement the epilog as illustrated in Figure 5b. The number of shifts required depends on the number of elements that are operated upon in the vector (`log2(nelements)`). This scheme requires introducing operation-codes for vector-shifts, and defining a new vector-shift operation in GCC. One approach we considered was to advertise two kinds of vector-shift operations. One that is more general and takes an arbitrary shift amount in bits. The second is the most basic vector-shift we can introduce - it takes a constnat (immediate value) shift amount in bytes. This has the potential to be easily expanded to the most efficient code, and be applicable to as many targets as possible: by supplying an API that would let the vectorizer ask for precisely the capability it needs (and all it needs here is a vector shift of a constant value in bytes), we let targets that don't support non-constant shifts, or that support only byte shifts, to also enjoy this feature. Examining the available target support seems to support this approach. Altivec has immediate byte-shifts, and less efficient non-immediate bit-shifts (because they require putting the scalar shift amount in a vector register first, an operation that is carried out through memory); SSE also has only immediate vector shifts. However, we decided to introduce only the general vector shift. One reason is that all other shifts that GCC supports take the shift count in bits, and we didn't want to create a confusion. Also, it is not the common practice in GCC to introduce an operation both in constant-argument form and non-constant-argument form. Most importantly, we do not sacrifice performance here by introducing only general shifts, since targets that can't convey that they support general vector shifts, but that do have immediate vector shifts, can instead advertize that they support the

(a) ia64: reduc_op epilog (sfp):

```
  mov f7 = f0
  addl r14 = 511, r0
  fpack f8 = f1, f1 ;;
  mov ar.lc = r14
loop:
  ldf8 f6 = [r14], 8 ;;
  fpma f7 = f7, f8, f6
  br.cloop.sptk.few loop ;;
epilog:
  fswap f6 = f7, f0 ;;
  fpma f6 = f7, f8, f6 ;;
extract_scalar:
  getf.sig r14 = f6 ;;
  setf.s f8 = r14
```

(b) Altivec: shifts epilog (sfp):

```
  li r0,64
  mtctr r0
  vxor v1,v1,v1
loop:
  lvx v0,r0,r9
  addi v9,v9,16
  vaddfp v1,v1,v0
  bdnz loop
epilog:
  vsldoi v0,v1,v1,8
  vaddfp v0,v1,v0
  vsldoi v1,v0,v0,12
  vaddfp v0,v0,v1
extract_scalar:
  addi r9,r1,28
  stvewx v0,r0,r9
  lfs f1,28(r1)
```

(c) SSE2: scalar epilog (dfp):

```
  xorl  %eax,%eax
  xorpd %xmm0,%xmm0
loop:
  addpd a(%eax,%eax),%xmm0
  addl  $8,%eax
  cmpl  $1024,%eax
  jne   loop
epilog:
  movapd %xmm0,%xmm1
  movsd  %xmm1,-8(%ebp)
  fldl   -8(%ebp)
  unpckhpd %xmm0,%xmm0
  movsd %xmm0,-8(%ebp)
  fldl  -8(%ebp)
  faddp %st, %st(1)
```

Figure 6. Reduction epilogs generated for a summation of floats

Domain	Name	Description	Data-Type	Features
linear algebra	saxpy	constant times a vector plus a vector	floats	
linear algebra	sdot	dot product of two vectors	floats	reduction
video	chromakey8,chromakey16	replace beackground with another image	chars,shorts	if-conversion
general	max_s16,max_u8	find maximum over elemnts of a vector	signed shorts, unsigned chars	reduction
general	sum_u8	summation of a vector elements	(unsigned) chars	reduction
video	sad8	sum of absolute differences	chars	reduction

Table 3. Benchmark Description

relevant `reduc_op` operation, by implementing it using immediate vector shifts in their machine description. So we can keep the GIMPLE IL general and clean, without loss of functionality or performance.

Scalar epilog. The last option we resort to, if no other vector support is available, is illustrated in Figure 5c - we sequentially compute the reduction on the scalar elements (the partial results) of the vector.

Figure 6 shows an example of a summation reduction that illustrates the different epilogs generated for different targets (we added labels in the assembly for readability). For this example we show a single precision floats reduction on ia64 and Altivec, and double precision floats reduction on SSE2. ia64 models the `reduc_op` idiom for this data type using a sequence of two instructions. Altivec uses the shifts epilog in this case. SSE2 uses a scalar epilog to sum-up the final partial results.

5 Experimental results

The results we present in this section were generated automatically using the GCC 4.1 compiler, and for features that have not yet been merged into the main GCC trunk we used the GCC autovect-branch compiler. Both compil-

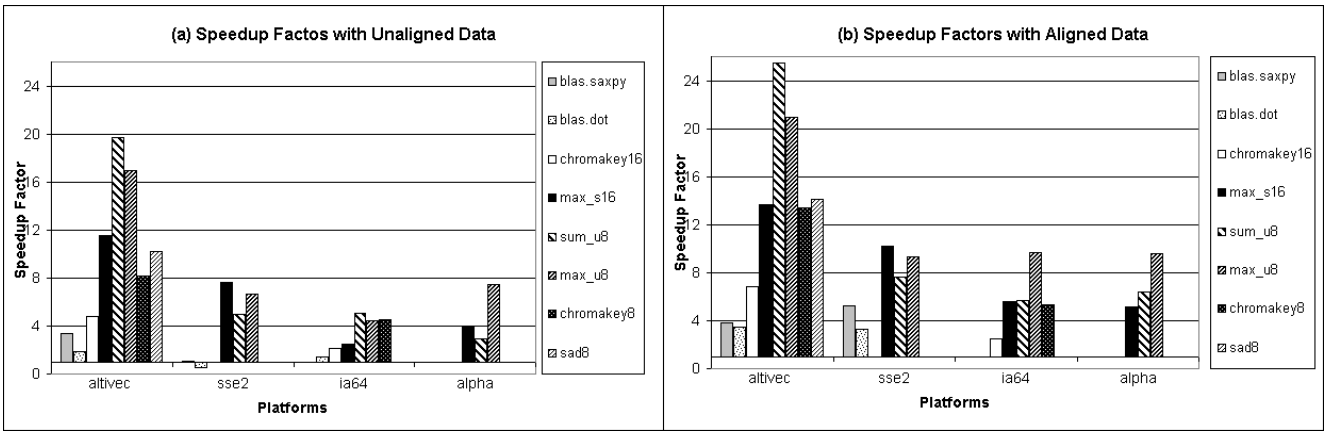

Figure 7. Autovectorization speedups across different platforms.

ers are available from [6]. For obtaining the experimental results we focused on 4 of the 8 platforms that GCC can currently vectorize for (see Table 1), as the other platforms were not available to us. Experiments were performed on: (1) IBM PowerPC PPC970 processor with Altivec support, (2) Intel Pentium4 processor with SSE2 support, (3) Itanium2, and (4) Alpha, all running under linux. Table 3 provides a brief description of the kernels used in our experiments. We report the speedup factors achieved by an automatically vectorized version over the sequential version of the benchmark, compiled with the same optimization flags. Time is measured using the `getrusage` routine, and includes any overheads introduced by vectorization. Figure 7a summarizes the speedup factors obtained on benchmarks in which the alignment of the data is unknown, and Figure 7b shows the speedups when the data is aligned.

PPC970 Speedups (Altivec): On PPC970 we would expect an improvement factor of 2 on the floating point (fp) computations (a 4-way SIMD unit vs. 2 scalar units) and an improvement factor of 4 on the fp loads and stores. The speedup factors obtained on the fp benchmarks are within that range, achieving 3.4/1.9 on sdot (aligned/unaligned respectively) and 3.8/3.4 on saxpy. saxpy is more memory intensive than sdot and therefore the higher speedups. The computations on short elements are expected to obtain a speedup between 4 (an 8-way SIMD unit vs. 2 scalar units) and 8 (on the loads and stores). Additional speedup is expected due to improved Instruction Level Parallelism (ILP), as the scalar integer unit manipulates the loop index in parallel to the computations in the vector unit. The speedups acheived are 6.9/4.8 on chromakey8 (aligned/unaligned) and 13.7/11.5 on max_s16. The super-linear speedups on max_s16 are because the `max` instruction is available on the vector unit, but not on the scalar unit. Finally, the kernels that operate on chars are expected to improve by a factor between 8 (a 16-way

SIMD unit vs. 2 scalar units) and 16 (on the loads and stores), with additional improvements due to enhanced ILP. The speedups we obtain are 25.2/19.7 on sum_u8 (aligned/unaligned), 20.91/16.99 on max_u8, 13.37/8.19 on chromakey8, and 14.11/10.18 on sad8. The super-linear improvements we observe on sad8 and max_u8 are due to the absence of a `max` and `abs` instructions on the scalar unit, while they are available on the vector unit. The super-linear improvement on sum_u8 is due to suboptimal code generation for the scalar version - an extra copy is generated that also creates dependencies that hinders subsequent optimizations.

Pentium4 Speedups (SSE2): While the SSE2 arhitecture supports 128bit vectors, the Pentium4 implementation can only operate on 64bit simultaneously. Therefore, the expected improvement on the fp kernels is around 2, minus alignment handling overhead on the unaligned versions. Additional improvements are expected due to the fact that the scalar fp computation takes place on the x87 fp stack, which is slower than the SSE2 unit. The speedups on the fp benchmarks are 3.32/0.52 on sdot (aligned/unaligned), and 5.25/1.04 on saxpy. Alignment handling is very inefficient on SSE2 because the software-pipelined scheme cannot be used here (as the alignments are unknown, and SSE2 cannot support realignment in this case). The kernels that operate on short elements should enjoy an improvement factor of 4 (because the Pentium4 operated on 64bit data at a time). We couldn't measure results on chromakey16 because conditional vector operations are not yet modeled in the SSE2 md files (this is expected to change in the near future). On max_s16 we achieve a speedup factor of 10.2/7.65 (aligned/unaligned). The super-linear speedup is due to inefficient code that is generated for the scalar version - the signed shorts are loaded with zero extension and then an extra copy with sign extension takes place. Indeed, if we check the speedups on a similar kernel operating on unsigned shorts we get the expected improvements - 4.64/4.73 (aligned/unaligned). The char kernels are expected to improve by a factor of 8, minus alignment handling overhead on the unaligned versions. The speedups we obtain are 7.66/4.98 (aligned/unaligned) on sum_u8 and 9.28/6.63 on max_u8, which are within the expected range, with additional improvement for max_u8 due to the availability of the `max` instructions in vector form only. Currently chromakey8 and sad8 do not get vectorized on SSE2 due to lack of modelling in the target md files (this is expected to be added in the near future).

Itanium2 speedups: We would expect an improvement factor of 2 for floats, as the SIMD float instructions run on the same function units as the scalar float instructions. For integer data, we would expect the improvement to be related to the number of elements packed in the word size.

The improvement factor for the fp cases are 0.98/1.38 (aligned/unaligned) for sdot and 2.06/2.21 for saxpy. Unfortunately, many of the measured results do not live up to expectations due to a failure in the induction variable

eliminator affecting 64-bit targets; this is believed to be fixed on a different source branch, but could not be merged to our working branch within time constraints. This is the reason for the low results in the aligned case for sdot. In the case of saxpy, the scalar code path was affected as well, so while the absolute performance is not as desired, the improvement factor remains as expected.

On the integer side we have 2.45/2.13 for chromakey16, 5.32/4.53 for chromakey8, 5.55/2.45 for max_s16, 9.67/4.44 for max_u8. Currently sad8 is not modelled in the target md file. In the case of chromakey, if-conversion on vectors is less efficient than on scalars; it is still valuable, but the results are only half linear improvement. In the case of max, the raw max operation is available to scalar code, so the only possible reason for the super-linear improvement is reduced overhead in the memory unit.

Alpha Speedups: As an older platform, Alpha does not have the breadth of SIMD instructions as on the other platforms. But for the data types supported, we do see improvements of 5.14/3.96 for max_s16 and 9.6/7.43 for max_u8. Note that while there is no vector addition instruction, there are some tricks that can be played with logical operations to implement it anyway. This provides a modest improvement of 1.65/1.31 for sum_u8.

6 Related Work

SIMD auto-vectorization capabilities have been incorporated in recent years in a number of research and industry compilers. These include works based on xlc (the IBM compiler) [22, 21, 4], icc (the Intel compiler) [2, 1], VAST [18], GCC [12, 5], the CoSy compiler framework [7], Chameleon (an IBM research compiler) [13], and numerous works based on the SUIF compiler [8, 9, 16, 17, 3, 19, 10].

The problem of handling data alignment is discussed in detail in [21, 4]. They present a technique to minimize the number of realignments necessary to satisfy the hardware alignment constraints. This technique is applicable to SIMD targets that can support a ”shiftstream” idiom, which is equivalent to a vector shift. They report results on an AltiVec platform. Alignment is also addressed in [9], but in the context of avoiding unaligned accesses through analyses and loop transformations. It reports results on an Altivec platform using the VAST compiler (by communicating the results of their SUIF analysis to VAST). Avoiding unaligned accesses is also the main focus of the alignment support for MMX/SSE discussed in [2, 1]. They can handle misaligned accesses, but without exploiting the data reuse across loop iterations. Our work addresses a larger range of alignment capabilities and discusses several alternatives for handling and representing alignment mechanisms. Like [1] we also incorporate peeling and versioning in our alignment handling framework, and like [21] we also handle run-time alignment

while exploiting reuse, but our work is less aggressive in minimizing realignment operations.

With respect to SIMD vectorization in general, much of the research work has been targeted at Altivec platforms. Among these are [22] that proposes a vectorization framework for SIMD that addresses alignment, mixed data types and multiple scopes for extracting parallelism. A straight-line-code vectorization technique called SLP (as opposed to the classic loop based approach) is proposed in [8], and extended by [16] to work in the presence of control flow. SLP is also used in [17], which presents a technique to reuse data in vector registers, treating them as a compiler controlled cache. These SLP based works all report results on Altivec platforms. The VAST compiler [18]. also targetted at Altivec, reports handling of reduction and alignment, but there's no public information available of their techniques.

Vectorization for the MMX extensions is reported in [19]. They use classic analyses to detect data parallel loops, enhanced by loop transformations. An extensive discussion of vectorization for MMX/SSE is provided in [2, 1].

Vectorization on VIS is demonstrated in [7], where they propose a vectorization technique based on loop unrolling . They also report handling of reduction, but do not handle unaligned accesses (they create a runtime alignment check instead). VIS is also the target platform of [3]. They present a two phase framework, that first vectorizes for infinite length vectors, and then transforms them into VIS instructions. They generate unaligned vector accesses, guarded by a run-time test, and also don't exploit data reuse.

The work reported in [10] is based on MIPS. They explore of tradeoffs between out-of-order short vectors and in-order large vectors using a simulator. Reduction and alignment are not addressed by this work. A SIMdD DSP platform that supports SIMD operations on disjoint data elements via vector pointers is addressed in [13]. Reduction is supported by this work, and alignment is handled as a special case of data reorganization using the vector pointers.

None of this prior art addresses the issues that come up when trying to vectorize for multiple SIMD platforms. A first step in this direction was presented in [12], but no experimental results were demonstrated in that work. This work is the first to our knowledge to target more than one SIMD target, demonstrating results on 4 different SIMD platforms using one vectorization framework.

7 Conclusions and Future Work

SIMD platforms are becoming pervasive across the industry. The hardware limitations that they exhibit, (e.g. data alignment restrictions) and unique architectural mechanisms that they provide require special care by a vectorizing compiler. In this paper we presented a high-level target-independent compilation technology that addresses these issues in a multi-platform setting. Our automatic vectorization scheme is able to balance the conflicting needs that arise from the diverse nature of SIMD architectures, while efficiently supporting each individual platform. This is to our knowledge the first effort that considers the multi-platform aspect of vectorization, and demonsrates experimental results on several different architectures using one compiler.

Following its incorporation into the GCC source base, the auto-vectorizer has received a lot of interest from users, applying it to different programs and platforms. We plan to continue support and address users' feedback, while continuing to enhance the vectorizer. One of the most important future directions is devising a cost model to better estimate when it is profitable to apply vectorization. Also, in order to increase benchmark coverage, we plan to add support for runtime aliasing tests, data-type conversions, and strided accesses.

8 Acknowledgments

We would like to thank the GCC community for their comments and feedback, and to those who contributions to the vectorizer - Ira Rosen, Olga Golovanevsky, Devang Patel and Keith Besaw.

References

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

[2] Aart Bik. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[3] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *Second SUIF Compiler Workshop*, August 1997.

[4] Alexandre E. Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, June 2004.

[5] Free Software Foundation. Auto-Vectorization in GCC, http://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[6] Free Software Foundation. GCC, http://gcc.gnu.org.

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

[8] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *PLDI*, 35(5):145–156, June 2000.

[9] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and detecting memory address congruence. In *PACT*, September 2002.

[10] Corinna G. Lee and Mark G. Stoodley. Simple vector microprocessors for multimedia applications. In *Proceesings of the International Symposium on Microarchitecture*, pages 25–36, 1998.

[11] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *the GCC Developer's summit*, June 2003.

[12] Dorit Naishlos. Autovectorization in gcc. In *the GCC Developer's summit*, pages 105–118, June 2004.

[13] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a simdd dsp architecture. In *CASES*, pages 2–11, October 2003.

[14] Diego Novillo. Tree ssa - a new optimization infrastructure for gcc. In *the GCC Developer's summit*, June 2003.

[15] 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.

[16] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO*, March 2005.

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

[18] Crecent Bay Software. VAST-F/ALtivec: Automatic Fortran Vectorizer for PowerPC Vector Unit, http://www.crescentbaysoftware.com/docs/vastfav.pdf.

[19] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–400, August 2000.

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

[21] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient simd code generation for runtime alignment. In *CGO*, March 2005.

[22] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *ICS*, June 2005.