Reprinted from the

Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Autovectorization in GCC

Dorit Naishlos
IBM Research Lab in Haifa
dorit@il.ibm.com

Abstract

Vectorization is an optimization technique that has traditionally targeted vector processors. The importance of this optimization has increased in recent years with the introduction of SIMD (single instruction multiple data) extensions to general purpose processors, and with the growing significance of applications that can benefit from this functionality. With the adoption of the new Tree SSA optimization framework, GCC is ready to take on the challenge of automatic vectorization. In this paper we describe the design and implementation of a loop-based vectorizer in GCC. We discuss the new issues that arise when vectorizing for SIMD extensions as opposed to traditional vectorization. We also present preliminary results and future work.

1 Introduction

Vector machines were introduced in the 1970's, to increase processor utilization by accelerating the initiation of operations, and keeping the instruction pipeline full. To take advantage of vector hardware, programs are rewritten using explicit vector operations on whole arrays (as in Figure 1b) instead of operations on individual array elements one after the other (as in Figure 1a). This rewrite of loops into vector form is referred to as *vectorization* [3].

The vector notation in Figure 1b implies that all the loads (from arrays a and b) take place before all the stores (into array a); This means that loops like the one in Figure 1d, where each iteration uses a result from a previous iteration, cannot be rewritten in vector form. This situation is an example of a data-dependence that is carried across the iterations of the loop. When no such dependences between loop iterations exist, operations from different iterations can be initiated in parallel, and vectorization may be applied. Data dependence analysis is therefore a fundamental step in the process of vectorization.

Vectorization, when applied automatically by a compiler, is referred to as *autovectorization*. In this paper, we use the two terms interchangeably to refer to compiler vectorization.

In recent years, a different architectural approach to exploit a similar kind of data parallelism has become increasingly common. It follows the Single Instruction Multiple Data (SIMD) model, in which the same instruction simultaneously executes on multiple data elements that are packed in advance in vector registers. The length of these vectors (*vector length*) is relatively small. The number of data elements that they can accommodate determines the degree of parallelism (*vectorization factor*, VF) that can be exploited. This value varies depending on the data-type of the elements.

Vectorizing the loop in Figure 1a for SIMD therefore implies transforming it to operate on VF elements at a time, as illustrated in Fig-

(a) original serial loop:

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

(b) loop in vector notation:

```
a[0:N] = a[0:N] + b[0:N];
```

(c) vectorized loop:

```
for (i=0; i<(N-N%VF); i+=VF){
  a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
}
for ( ; i < N; i++) {
  a[i] = a[i] + b[i];
}
```

(d) unvectorizable loop (dependence cycle):

```
for (i=1; i<N; i++){
  a[i] = a[i-1] + b[i];
}
```

Figure 1: The vectorization transformation

ure 1c. This is generally equivalent to strip-mining the loops by a factor `VF`, while replacing scalar operations with equivalent vector operations. A serial loop that computes the remaining `N%VF` iterations is also added for the case that `N` does not evenly divide by `VF`.

Applications in many domains have an abundant amount of natural parallelism present in the computations they perform. If this parallelism can be leveraged to exploit the SIMD/vector capabilities of architectures, the performance of these applications can be considerably increased.

The GCC vectorizer implements a loop-based vectorization approach, which means that it focuses on exploiting the data parallelism present across loop iterations. Data parallelism present in straight-line code is not leveraged by the loop-based vectorizer. Vectorization techniques that exploit this type of parallelism, such as [12], could be used as a complementary approach to loop-based vectorization. We briefly discuss this in Section 8.

As we show in the following sections, practical vectorization for vector processors, and in particular the more recent SIMD processors, involves much more than loop dependence analysis and introduces some nontrivial issues and choices especially for a multi-platform compiler like GCC.

2 Classic Vectorization vs. SIMD Vectorization

Autovectorization is a mature research area; automatic detection of vector loops in serial code has been discussed in literature for more than a decade [1, 20]. The main focus of classic vectorization is the theory of data dependences. It deals with loop analyses to (1) detect statements that could be executed in parallel without violating the semantics of the program, and (2) increase such occurrences by means of loop transformations.

The classic (data-dependence based) vectorization approach has traditionally targeted the vector machines of the 1970's. Two main developments in recent years have shifted the focus to vectorizing for the modern SIMD architectures. One is the proliferation of SIMD capabilities in modern computing platforms, including gaming machines [18], Digital Signal Processors (DSPs) [7, 10], and even in general purpose processors [15, 8]. The second factor is the growing significance of applications that can benefit from SIMD functionality, particularly those in the multimedia domain.

The classic vectorization theory does not apply very successfully to SIMD machines [16], for several reasons. Traditional vectorization has focused on array-based Fortran programs from the scientific computing domain. Many of the important modern workloads, such as multimedia applications, are written in C and make extensive use of pointers. The presence

of pointers, and other programming language differences [2] give rise to a new domain of problems critical for the success of vectorization.

The primary difficulties in applying classic vectorization to SIMD lie in the architectural differences between SIMD extensions and traditional vector architectures. First, SIMD memory architectures are typically weaker than those of traditional vector machines. The former generally only support accesses to contiguous memory items, and only on vector-length aligned boundaries. Computations, however, may access data elements in an order which is neither contiguous nor adequately aligned. SIMD architectures usually provide mechanisms to reorganize data elements in vector registers in order to deal with such situations. These mechanisms (packing and unpacking data elements in and out of vector registers and special permute instructions) are not easy to use, and incur considerable penalties. For a vectorizer, this implies that generating vector memory accesses becomes much more involved.

In addition, the instruction sets of SIMD architectures tend to be much less general purpose and less uniform. Many specialized domain-specific operations are included, many operations are available only for some data types but not for others, and often a high-level understanding of the computation is required in order to take advantage of some of the functionality. Furthermore, these particular characteristics differ from one architecture to another.

These attributes demand that low-level architecture-specific factors will be considered throughout the process of vectorization. Classic dependence analysis is therefore only a partial solution to vectorizing for SIMD extensions. Code transformation issues require much more attention, as discussed later in the paper.

3 Data-Dependence Analysis

The classic approach for vectorization is based on the theory of data-dependence analysis. Some parts of the classic data-dependence-based analyses and transformations are already present in GCC (currently in the loop-nest-optimizations (lno) branch of GCC).

The first step in dependence analysis is the construction of a data dependence graph (ddg). The nodes of the graph are the loop statements, and edges between statements represent a data dependence between them. There are two types of such edges. Edges between scalar variables represent a def-use link between statements. These links can be trivially computed from a SSA representation, such as the one used in the tree-ssa representation level of GCC.

The second kind of edges are those between memory references. The classic data-dependence theory focused on array-based Fortran programs, and therefore only array references have traditionally been considered. In other (e.g., C) programs, memory references can take other forms (e.g., indirect references through pointers), and these are considered by the GCC vectorizer. Memory dependences are determined by applying a set of dependence tests [9, 3] that compare array subscripts. Simpler and faster tests (GCD, Banerjee) are applied to simple subscript forms. More complex and accurate tests (Gamma, Delta, Omega) are applied to more complicated subscripts.

If a dependence is carried by the relevant nesting level then an edge is added to the ddg. For example, in Figure 1, loops (a) and (d) both have a dependence between the two references to array `a`, but only the dependence in loop (d) is carried across the loop itera-

tions and prevents vectorization. In GCC, tests that compute dependences between array references are implemented in the module `tree-data-refs.c`. They are used by the vectorizer to detect dependences between data references in inner-most loops.

The classic data dependence analysis proceeds to detect Strongly Connected Components (SCCs) in the ddg. SCCs that consist of a single node represent a statement that can be executed in parallel at the loop level that is being considered. SCCs that consist of multiple nodes represent statements that are involved in a dependence cycle, and prevent the vectorization of the loop unless the cycle can be "broken."

In order to increase the potential for vectorization, the vectorizable parts can be separated from the groups of statements that are involved in dependence cycles (loop distribution). This is done by creating a separate loop for each SCC, after having topologically sorted the reduced graph in which each SCC is represented by a single node. There is a preliminary implementation of ddg construction in GCC (as part of the scalar-evolutions module) but it is not yet used. Loop distribution to increase vectorization opportunities is not yet supported, however other loop transformations that increase parallelism (loop interchange, scaling, skewing, and reversal) are supported in GCC as part of the tree-loop-linear module.

Lastly, special kinds of dependence cycles can be dealt with if recognized as certain idioms, such as reduction. The GCC vectorizer will be enhanced to recognize and handle such situations in the near future.

4 Vectorizer Overview

The vectorization optimization pass is developed in the loop-nest-optimizations (lno) branch (`http://gcc.gnu.org/projects/tree-ssa/lno.html`), at the IR level of SSA-ed GIMPLE trees. The current development status can be found on `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.
The vectorizer is enabled by the `-ftree-vectorize` flag which also invokes the scalar-evolutions analyzer, upon which the vectorizer relies for induction variable recognition and loop bound calculation. The bulk of the vectorizer functionality can be found in two files (`tree-vectorizer.c` and `tree-vectorizer.h`). The vectorizer also uses loop-related utilities that reside elsewhere, many of which are new contributions developed in the lno branch.

The vectorization pass is in early stages of development; the basic infrastructure is in place, supporting initial vectorization capabilities. These capabilities are demonstrated by the vectorization test-cases, which are updated to reflect new capabilities as they are added. Work is underway to extend these capabilities and to introduce more advanced vectorization features.

4.1 vectorizer layout

An outline of the vectorization pass is given in Figure 2. The main entry to the vectorizer is `vectorize_loops(loops)`. The vectorizer applies a set of analyses on each loop, followed by the actual vector transformation for the loops that had successfully passed the analysis phase.

4.2 vectorizer analysis

The first analysis phase, `analyze_loop_form()`, examines the loop exit condition and number of iterations, as well as some control-flow attributes such as number of basic blocks and nesting level. One major restriction im-

```
vect_analyze_loop (struct loop *loop) {
  loop_vec_info loopinfo;
  loop_vinfo = vect_analyze_loop_form (loop);
  if (!loop_vinfo)
    FAIL;
  if (!analyze_data_refs (loopinfo))
    FAIL;
  if (!analyze_scalar_cycles (loopinfo))
    FAIL;
  if (!analyze_data_ref_dependences (loopinfo))
    FAIL;
  if (!analyze_data_ref_accesses (loopinfo))
    FAIL;
  if (!analyze_data_refs_alignment (loopinfo))
    FAIL;
  if (!analyze_operations (loopinfo))
    FAIL;
  LOOP_VINFO_VECTORIZABLE_P (loopinfo) = 1;
  return loopinfo;
}

vect_transform_loop (struct loop *loop) {
  FOR_ALL_STMTS_IN_LOOP(loop, stmt)
    vect_transform_stmt (stmt);
  vect_transform_loop_bound (loop);
}
```

Figure 2: Vectorizer outline

(a) uncountable loop:

```
while (*p != NULL){
  *p++ = X;
}
```

(b) reduction - summation:

```
for (i=0; i<n; i++){
  sum += a[i];
}
```

(c) induction:

```
for (i=0,j=0; i<n; j++,i++){
  a[i] = j;
}
```

(d) non consecutive access pattern:

```
for (i=0; i<n; i++){
  a[2*i] = X;
}
```

Figure 3: Loop examples

posed on a loop for vectorization to be applicable, is that the loop is countable—i.e, an expression that calculates the loop bound can be constructed and evaluated either at compile time or at run-time. For example, the loop in Figure 3a is not a countable loop. The loop bound analysis is carried out by the scalar evolution analyzer. To simplify the initial implementation, the vectorizer also verifies that the loop is an inner-most loop, and consists of a single basic block. Multi-basic-block constructs such as if-then-else are collapsed into conditional operations if possible, by an if-conversion pass prior to vectorization.

Next, `analyze_data_refs()` finds all the memory references in the loop, and checks if they are "analyzable"—i.e., an access function that describes their modification in the loop (evolution) can be constructed. This is required for the memory-dependence, access-pattern and alignment analyses (described in Section 5). Other restrictions enforced at this point are there for simplicity of implementation, and will be relaxed in the near future.

Dependences which do not involve memory operations are analyzed directly from the SSA representation. The function `analyze_scalar_cycles()` examines such "scalar-cycles" (dependence cycles which involve only scalar variables), and verifies that any scalar cycle, if exists, can be handled in a way that breaks the cross-iteration dependence.

One kind of such "breakable" scalar cycles are those that represent reductions. A reduction operation computes a scalar result from a set of data-elements. The loop in Figure 3b for example, computes the sum of a set of array elements into a scalar result `sum`. Some reduction operations can be vectorized, generally by computing several partial results in parallel, and combining them at the end (reducing them) to single result. Scalar cycles can also be created by induction variables (IVs). Certain IVs that are used for loop control and for address computation, are handled as an inherent part of vectorization. An example of an IV of this type

is `i` from Figure 3c. Other IVs such as `j` from the same example require special vectorization support. Support for vectorization of reduction and induction will be introduced to GCC in the near future.

The final analysis phase `analyze_operations()` scans all the operations in the loop and determines a vectorization factor. The vectorization factor (VF) represents the number of data elements that will be packed together in a vector, and is also the strip-mining factor of the loop. It is determined according to the data-types operated on in the loop, and the length of the vectors supported by the target platform. Currently we use a simple approach that allows a single vector length per platform and a single data-type per loop, but these restrictions will be relaxed in the near future. `analyze_operations()` also checks that all the operations can be supported in vector form. The cost of expanding them to scalar code in case they are not supported, is expected to offset the benefits of vectorizing the loop. In the future, a cost model should be devised to support the vectorizers decisions on which loops to vectorize.

4.3 vectorizer transformation scheme

During the analysis phase the vectorizer records information at three levels of granularity—at the loop level (`loop_vect_info`), at the statement level (`stmt_vec_info`), and per memory reference (`data_reference`). These data-structures are later used during the loop transformation phase.

The vectorization transformation can be generally described as "strip-mine by VF and substitute one-to-one," which implies that each scalar operation in the loop is replaced by its vector counterpart. The loop transformation phase scans all the statements of the loop top-down (defs are vectorized before their uses), inserting a vector statement VS in the loop for each scalar statement S that needs to be vectorized, and recording in the `stmt_vec_info` attached to S a pointer to VS; This pointer is used to locate the vectorized version of statement S during the vectorization of subsequent statements that depend on S.

(a) before vectorization:

```
S1:  x = b[i];
S2:  z = x + y;
S3:  a[i] = z;
```

(b) after vectorization of S1:

```
VS1: vx = vpb[indx];
S1:  x = b[i];   ---> VS1
S2:  z = x + y;
S3:  a[i] = z;
```

(c) after vectorization of S2:

```
VS1: vx = vpb[indx];
S1:  x = b[i];   ---> VS1
VS2: vz = vx + vy;
S2:  z = x + y;  ---> VS2
S3:  a[i] = z;
```

(d) after vectorization of S3:

```
VS1: vx = vpb[indx];
S1:  x = b[i];   ---> VS1
VS2: vz = vx + vy;
S2:  z = x + y;  ---> VS2
VS3: vpa[indx] = vz;
```

Figure 4: The transformation process

After all statements have been vectorized, the original scalar statements may be removed. Stores to memory are explicitly removed by the vectorizer; the remaining scalar statements are expected to be removed by dead code elimination pass after vectorization.

Figure 4 illustrates the transformation process; First, stmt S1 is vectorized into stmt VS1; In order to vectorize stmt S2, the vectorizer needs to find the relevant vector def-stmt for each operand of S2. The figure only shows how

this is done for the operand x: first, the scalar def-stmt of x, S1, is found (using SSA), and the relevant vector def vx is retrieved from the vectorized statement of S1, VS1. Similarly for S3, except that S3 is also removed.

In practice, many cases require additional handling beyond the "one-to-one substitution." Constants and loop invariants require that vectors be initialized at the loop pre-header. Other computations require special epilog code after the loop (e.g., reductions, and inductions that are used after the loop). Some access-patterns require special data manipulations between vectors within the loop (e.g., data interleaving and permutations). Some scalar operations cannot be replaced by a single vector operation (e.g., when mixed data-types are present). Sometimes a sequence of scalar operations can be replaced by a single vector operation (e.g., saturation, and other special idioms). It might be possible to hide some of these complications from the vectorizer, and handle them at lower levels of code generation. We discuss these issues in Section 6.

Finally, the loop bound is transformed to reflect the new number of iterations, and if necessary, an epilog scalar loop is created to handle cases of loop bounds which do not divide by the vectorization factor. This epilog also must be generated in cases where the loop bound is not known at compile time.

5 Handling Memory References

Memory references require special attention when vectorizing. This is true in the classic vectorization framework, and even more so when vectorizing for SIMD. The vectorizer currently considers two forms of data-refs—one-dimensional arrays (represented as ARRAY_REFs that are VAR_DECLs), and pointer accesses (INDIRECT_REFs). Once an access function has been computed (for the array index or the pointer) the vectorizer proceeds to apply a set of data-ref analyses, which we describe here.

5.1 Dependences and aliasing

As mentioned above, one of the basic restrictions that has to be enforced in order to safely apply vectorization is that no dependence cycles exist. A simplified form of the standard memory dependence analysis, which we briefly described in Section 3, is applied, using simple dependence tests from the tree-data-ref module of the lno-branch.

This analysis can be enhanced in several directions, including: (1) using more complex dependence tests, (2) pruning dependences with distance greater than the vectorization factor, and (3) not failing when a dependence is found, but instead attempting to resolve the dependence by reordering nodes in the dependence graph (consequently distributing the loop).

Pointer accesses require in addition alias analysis to conclude whether any two pointer-accesses in the loop may alias. If we cannot rule out the possibility that two pointers may alias, loop versioning can be used, with a run-time-overlap test to guard the vectorized version of the loop.

5.2 Access pattern

When the data is laid out in memory exactly in the order in which it is needed for the computation, it can be vectorized using the simple one-to-one vectorization scheme. However, computations may access data elements in an order different from the way they are organized in memory. For example, the computation in Figure 3d uses a strided access pattern (with stride 2). Non-consecutive access patterns usually require special data manipulations to re-

order the data elements and pack them into vectors. This is because the memory architecture restricts vector data accesses to consecutive vector-size elements.

Some architectures provide relatively flexible mechanisms to perform such data manipulations (gather/scatter operations in traditional vector machines, indirect access through vector pointers [14]). SIMD extensions usually provide mechanisms to pack data from two vectors into one (and vice versa), while possibly applying a permutation on the data elements. Some SIMD extensions provide specialized support for certain access-patterns (most commonly for accessing odd/even elements for computations on complex numbers), but these are usually limited only to a few operations and a few data types.

The underlying data reorganization support determines whether vectorization can be applied, and at what cost. These data manipulations need to be applied in each iteration of the loop and therefore incur considerable overhead. In fact, some access patterns, such as indirect access, cannot be vectorized efficiently on most SIMD/vector architectures. The function `analyze_access_pattern()` verifies that the access pattern of all the data references in the loop is supported by the vectorizer, which is currently limited to consecutive accesses only.

5.3 alignment

Accessing a block of memory from a location which is not aligned on a natural vector-size boundary is often prohibited or bears a heavy performance penalty. These memory alignment constraints raise problems that can be handled 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 in each iteration of the loop.

In order to avoid these penalties, techniques like loop peeling and static and dynamic alignment detection [11, 13, 4] can be used. Alignment handling therefore consists of three levels: (1) static alignment analysis, (2) transformations to force alignment, including runtime checks, and (3) efficient vectorization of the remaining misaligned accesses.

The functions `compute_data_refs_alignment()` and `enhance_data_refs_alignment()` (called from `analyze_data_refs_alignment()`) are responsible for items (1) and (2) above. `compute_data_refs_alignment()` computes misalignment information for all data-references; currently only a trivial conservative implementation is provided.

Following the alignment computation, the function `enhance_data_refs_alignment()` uses loop versioning and loop peeling in order to force the alignment of data references in the loop. Loop peeling can only force the alignment of a single data reference, so the vectorizer needs to choose which data reference DR to peel for. In the peeled loop, only the access DR is guaranteed to be aligned. Loop versioning could be applied on top of peeling, to create one loop in which all accesses are aligned, and another loop in which only the access DR is guaranteed to be aligned. A cost model should be devised to guide the vectorizer as to which access to peel for, and whether to apply peeling or versioning or a combination of the two, considering the code size and runtime penalties. Figure 5 illustrates these different alternatives.

If data-references which are not known to be aligned still remain after `enhance_data_refs_alignment()`, the vectorizer will proceed to vectorize the loop only if the target platform provides mechanisms to support

(a) original loop, before alignment analysis:

```
for (i = 0; i < N; i++) {
 x = q[i]; //misalign(q) = unknown
 p[i] = y; //misalign(p) = unknown
}
```

(b) after compute_data_refs_alignment():

```
for (i = 0; i < N; i++) {
 x = q[i]; //misalign(q) = 3
 p[i] = y; //misalign(p) = unknown
}
```

(c) option 1—loop versioning:

```
if (p is aligned) {
 for (i = 0; i < N; i++) {
   x = q[i]; //misalign(q) = 3
   p[i] = y; //misalign(p) = 0
 }
} else {
 for (i = 0; i < N; i++) {
   x = q[i]; //misalign(q) = 3
   p[i] = y; //misalign(p) = unknown
 }
}
```

(d) option 2—peeling for access q[i]:

```
for (i = 0; i < 3; i++) {
 x = q[i];
 p[i] = y;
}
for (i = 3; i < N; i++) {
 x = q[i]; //misalign(q) = 0
 p[i] = y; //misalign(p) = unknown
}
```

(e) option 3—peeling and versioning:

```
for (i = 0; i < 3; i++) {
 x = q[i];
 p[i] = y;
}
if (p is aligned) {
 for (i = 3; i < N; i++) {
   x = q[i]; //misalign(q) = 0
   p[i] = y; //misalign(p) = 0
 }
} else {
 for (i = 3; i < N; i++) {
   x = q[i]; //misalign(q) = 0
   p[i] = y; //misalign(p) = unknown
 }
}
```

Figure 5: Alternatives for forcing alignment

misaligned accesses. Figure 6c presents a possible scheme for handling misalignment [6]. It relies on a pair of target hooks: one that calculates the misalignment amount, and represents it in a form that the second hook can use (a shift amount or a permutation mask). The second hook combines data from two vectors, permuted according to the misalignment shift amount. In some cases the code could be further optimized by exploiting the data reuse across loop iterations, as shown in Figure 6d.

Targets that support misaligned accesses directly, do not need to implement these hooks; in this case, misaligned vector accesses will look just like regular aligned vector accesses, as in Figure 6b. Section 6 discusses the tradeoffs involved in this implementation scheme.

6 Vectorization issues

An issue that repeatedly comes up during the development of the GCC vectorization is the tension between two conflicting needs. One is the requirement to maintain a high-level, platform-independent program representation. The other is the need to consider platform-specific issues and express low-level constructs during the process of vectorization.

In many ways, the tree-level is the suitable place for the implementation of a loop-based vectorizer in GCC. Arrays and other language constructs are represented in a relatively high-level form, a fact that simplifies analyses such as alignment, aliasing and loop-level data-dependences. Analyses are further simplified due to the SSA representation. Implementing the vectorizer at the tree-ssa level allows it to benefit from the vast suite of SSA optimizations, and in particular, the loop related utilities developed in the lno-branch.

On the other hand, at this IR level it is not so trivial to handle situations in which target-

(a) scalar data-ref:

```
    i = init;
LOOP:
    x = a[i];
    i++
```

(b) vectorized data-ref:

```
    vector *vpx = &a[init];
    indx = 0;
LOOP:
    vector vx = (*vpx)[indx];
    indx++
```

(c) vectorized data-ref with misalignment support:

```
    vector *vpx1 = &a[init];
    vector *vpx2 = &a[init+VF-1];
    shft = target_hook_get_shft
                          (&a[init])
    indx = 0;
LOOP:
    vx1 = (*vpx1)[indx];
    vx2 = (*vpx2)[indx];
    vx = target_hook_combine_by_shft
                   (vx1, vx2, shft)
    indx++;
```

(d) optimized misalignment support:

```
    vector *vpx1 = &a[init];
    vector *vpx2 = &a[init+VF-1];
    shft = target_hook_get_shft
                          (&a[init]);
    indx = 0;
    vx1 = (*vpx1)[indx];
LOOP:
    vx2 = (*vpx2)[indx];
    vx = target_hook_combine
                   (vx1, vx2, shft);
    indx++;
    vx1 = vx2;
```

Figure 6: Handling data-refs (load example)

specific information needs to be consulted, and even less trivial to handle situations in which target-specific constructs need to be expressed.

Misalignment is an excellent example of such a situation. The low-level functionality that supports misaligned accesses must somehow be expressed in the tree IR. The implementation should maintain the following properties: (1) It should hide low-level details as much as possible. (2) It should be general enough to be applicable to any platform. SIMD extensions vary greatly from platform to platform. (3) Despite these restrictions, it should be as efficient as possible on each platform.

In terms of the above criteria, the misalignment scheme that was presented in the previous section: (1) exposes the vectorizer to low-level details of misalignment support, (2) might not be general enough (it assumes that low-order address bits are ignored by load operations), and (3) is potentially inefficient for targets that would be better supported by alternative methods.

To tackle these problems, two alternatives can be considered. Alternative 1: Annotate misaligned accesses and let the subsequent RTL expansion pass handle the details. This is the most natural way to address architecture specific details. However, this solution can potentially be very inefficient, because it neglects to take advantage of data reuse between iterations. To do that, the lower-level RTL passes would have to rediscover the kind of loop-level information the vectorizer already had. Alternative 2: Hide all these implementation details in a "black box" target hook, that would generate the most efficient code for its platform. A disadvantage would be that functionality that is common to many targets would have to be duplicated. Also, the vectorizer would be unaware of what's going on, and would have difficulty estimating the overall cost of applying

vectorization, for example.

Low-level architectural details are not only problematic with respect to representing them at a high-level platform-independent abstraction. Specific architectural vector support can directly affect the vectorization transformation, and even determine whether it should be applied at all. These details must be considered during vectorization because the choices made at the vectorization stage are not easily altered at later low-level stages of compilation. This is especially true in cases of architectural features that require recognition of an entire computational idiom, a task best supported by high-level analysis (reductions for example may be difficult to identify without the entire context of the loop).

These are some of tradeoffs and decisions involved in the implementation of the GCC vectorizer. These kinds of problems often come up in optimizing compilers, but are especially evident in the context of SIMD vectorization, and even more so when implemented in a multi-platform compiler like GCC.

7 Status

The first implementation of a basic vectorizer in GCC was contributed to the lno-branch on January 1st, 2004. It has since been enhanced with additional capabilities, including support for vectorization of constants, loop invariants, and unary and bitwise operations. The vector test-suite (`gcc/gcc/testsuite/gdd.dg/tree-ssa-vect/`) reflects the current vectorization functionality. The domain of vectorizable loops can be summarized in terms of the supportable (1) loop forms, (2) data references, and (3) operations. Currently support includes (1) inner-most, single-basic-block loop forms, with a known loop bound divisible by the vectorization factor; (2) consecutive array data references for which alignment can be forced, and (3) operations that do not create a scalar-cycle (no reduction or induction), that all operate on the same data-type, and that have a vector form that can be expressed using existing tree-codes.

Recent development has focused on broadening the range of loop-forms and data references that the vectorizer can support. This includes the vectorization of loops with unknown loop bounds, an if-conversion pass that allows the vectorizer to handle some forms of multi-basic-block loops, vectorization of unaligned loads, and vectorization of pointer accesses. These features are likely to be added by the time this paper is presented, and will soon be followed by support for peeling and versioning for alignment. Other future directions include support for multiple data-types, and for reduction and induction operations. In the next section we discuss additional directions for further development of the vectorizer.

8 Future Work

Following is a list of potential enhancements to the vectorizer, organized into four categories:

Support additional loop forms. Support for unknown loop bounds and if-then-else constructs is nearly complete. The major remaining restriction on loop form is the nesting level. Vectorization of nested loops will be considered in the future.

Support additional forms of data references. Potential extensions in this category include enhancements to the dependence tests (as discussed in Section 5) and support for additional access patterns (reverse access, and accesses that require data manipulations like strided or permuted accesses). Exploiting data reuse as in [17] is an optimization related to data references that we plan to consider in the future.

Support additional operations. Vectorization of loops with multiple data-types and type casting is the first extension expected in this category. This capability requires support for data packing and unpacking, which breaks out of the one-to-one substitution scheme, and cannot be directly expressed using existing tree-codes. The next capabilities to be introduced will be support for vectorization of induction, reduction, and special idioms (such as saturation, min/max, dot product, etc.), using target hooks or adding new tree-code as necessary.

Other enhancements and optimizations. Two general capabilities that we are planning to introduce are support for multiple vector lengths for a single target, and the ability to evaluate the cost of applying vectorization. This will require some form of cost modelling for the vector operations. Interaction with other optimization passes should also be examined, and in particular, potential interaction with other (new) passes that might also exploit data parallelism. One example could be loop parallelization (using threads). Another example could be straight-line code vectorization (as opposed to loop based), such as SLP [12].

SLP is in many ways suitable for lower representation levels, as it analyzes addresses, and operates like a peep-hole optimization on a single basic block at a time. This is what gives SLP it's main strength—scalar operations are grouped together into a vector operation without a need to prove general attributes about an enclosing loop as a whole. (In fact, it is not even aware of any enclosing loops). This property allows it to vectorize code sequences that the loop based vectorizer does not target. However, this is also its main limiting factor, and it can benefit from loop-level information which is already available to the tree-level loop-based vectorizer. We are therefore considering implementing SLP at the tree-level, as a complementary solution to the loop-based vectorizer.

With the introduction of support for unknown loop bounds, pointers, misalignment, and conditional code, the GCC vectorizer will be in relatively good shape compared to other vectorizing compilers. The major remaining restrictions (inner-most loops, consecutive accesses and a single data-type per loop) tend to be common to vectorizing compilers in general [5, 19]. However, as the (long) list above implies, most of the exciting features still lie ahead.

9 Acknowledgments

The vectorizer directly uses, or otherwise benefits from, utilities developed in the lno-branch contributed by Sebastian Pop, Zdenek Dvorak, Devang Patel, and Daniel Berlin. Many thanks to Sebastian for continuious support for smooth interaction of his analyzer with the vectorizer, and to Zdenek for ongoing resposive maintenance of the lno-branch. I would like to thank Olga Golovanevsky for her contributions to vectorizer functionality, to Ayal Zaks and the IBM Haifa team for helpful discussions, and to GCC contributors who offered help, comments and patches.

References

[1] John Randal Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, October 1987.

[2] Randy Allen and Steve Johnson. Compiling c for vectorization, parallelization, an inline expansion. In *SIGPLAN Conference on Programming Languages Design and Implementation*, June 1988.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern*

Architectures—A dependence-based approach. Morgan Kaufmann, 2001.

[4] 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.

[5] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Ximmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.

[6] Apple Computer. `http://developer.apple.com/hardware/ve/`.

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

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

[9] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, June 1991.

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

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

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

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

[14] Jaime H. Moreno, V. Zyuban, U. Shvadron, F. Neeser, J. Derby, M. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. Asaad, T. Fox, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research and Development*, March 2003.

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

[16] Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. *16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.

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

[18] Sony. `http://www.us.playstation.com/`.

[19] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extension architectures. *International Symposium on Microarchitecture*, pages 25–36, 1998.

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