



Crescent Bay Software

VAST-C/Altivec

Automatic C/C++ Vectorizer for PowerPC Altivec/VMX

July 2004

Version 1.6

Preface

Document Number V99081: VAST-C/AltiVec User's Guide

This is a guide to the use of VAST-C/AltiVec. VAST-C/AltiVec is an automatic vectorizer for C and C++ programs on the PowerPC vector processors known as Motorola AltiVec and IBM VMX. This manual is designed to give programmers an understanding of VAST-C/AltiVec's capabilities and effective use.

Revision Record

Edition	Date	Description
1.0	8/99	First release of document in this form.
1.1	12/99	Added new -V switches.
1.2	3/00	Added -Vmessages switch to get vectorization info.
1.3	6/00	Added DEEP information, minor changes.
1.4	10/01	Minor corrections to pragmas, messages.
1.5	9/03	Minor corrections.
1.6	7/04	Minor general revisions, command line changes.

Copyright (C) 1999-2004, Crescent Bay Software. No unauthorized use or duplication is permitted. All rights reserved.

VAST and DEEP are registered trademark of Crescent Bay Software Corp..

AltiVec is a trademark of Motorola, Inc.

Crescent Bay Software, 10950 Washington Blvd., Suite 230, Culver City CA 90232, USA. Fax: (310)-836-7313 Phone: (310)-836-5183.

Email: info4@crescentbaysoftware.com.

Web: <http://www.crescentbaysoftware.com>.

Table of Contents

Preface	i
1. Introduction	1
Overview	1
VAST-C/AltiVec operation	1
User interaction	2
Performance tips.....	2
Optimizations	3
How to use this guide.....	3
Optional features.....	3
2. Invoking VAST-C/AltiVec	4
vcc driver.....	4
Usage	4
File extensions	4
vcc options	5
Example.....	6
Assertion levels.....	6
Notes on assertion levels.....	6
VAST-C/AltiVec switches	7
VAST-C/AltiVec options	8
3. User pragmas and messages	10
Overview	10
Pragma format	10
VAST pragma summary.....	11
Transformation pragmas.....	12
aligned.....	12

novector/vector.....	12
skip.....	12
noassoc/assoc	12
select	12
Data dependency pragmas	12
nodepchk/depchk.....	13
permutation	13
relation	13
DISJOINT	13
Vectorization messages	13
Diagnostic messages	14
Optimization inhibition messages	14
Informative messages	14
Error messages	14
4. Altivec Code Generation	15
Overview	15
Altivec code generation overview	15
Vector code generation sections	16
Vector code generation example	16
Alignment.....	18
Iteration count cleanup.....	19
Array reductions.....	19
Vector strip unrolling.....	19
Non-contiguous vectors	19
5. Optimization overview	21
Pointers and array references	21
Understanding loop variables	22
Loop types	22
"for" loops.....	22
"while" loops.....	23
"do" loops	23
Allowed statements.....	23
Iteration count.....	24
Structures.....	25
Assignment operators.....	25

Translation diagnostics on loops.....	26
6. Loop Optimizations	27
Overview	27
Loop fusion	27
Loop re-rolling.....	28
Loop unrolling	28
"while" to "for"	30
Common Subexpression Enhancement.....	30
Scalar division removal	31
Loop interchange.....	31
Loop nest collapse	32
Outer loop unrolling.....	32
7. Data dependency analysis	34
Overview	34
Data dependency examples	34
Function arguments	35
Pointers and data dependency	36
Potential dependency run-time testing	37
Ambiguous subscript resolution.....	38
Data dependency pragmas	38
nodepchk pragma.....	38
disjoint pragma.....	39
relation -- specifying relations between variables.....	39
permutation -- declaring safe indirect addressing	40
Data dependency conflict messages	40
8. Decision processes	43
Allowable conditional constructs.....	43
Two types of conditions	43
Removal of invariant IFs	44
Messages about conditionals	45
9. Inline expansion	46
Introduction	46

Inline expansion pragmas	47
autoexpand pragma	47
Explicit mode	47
expand pragma	47
nexpand pragma	48
search pragma	48
Where to get the code	48
Same file	48
Explicitly named file	49
Implicitly named file	49
Possible problems	49
Separate compilation	49
Code size	49
Debugging	49
Compilation rate	50
Analysis inhibitors	50
Expansion inhibitors	50
Inhibitors specific to automatic expansion mode	50
Inline expansion user messages	50
Inline expansion summary	50
User messages	51

Index

53

1. Introduction

Overview

This document describes the features and options of VAST-C/AltiVec, a software tool that vectorizes C programs for execution on AltiVec systems. VAST-C/AltiVec automatically transforms loops into vector instructions that use the AltiVec unit, which can result in substantially faster code.

Automatic vectorization of existing C programs is a very useful tool, but additional tuning of the generated code can sometimes be helpful in getting full performance from the system. VAST-C/AltiVec can generate varying degrees of vectorization in many programs, depending on the algorithms used, coding style, size of arrays, data types, and other factors.

VAST-C/AltiVec operation

To use VAST, you normally invoke the VAST driver appropriate for the compiler you're using, instead of the compiler itself. For example, for gcc the driver is **vcc**, and for IBM's xlc, the driver is **vlc**. However for brevity the text in this document usually refers just to "vcc".

These are the steps vcc performs:

1. Applies VAST-C/AltiVec to the input .c files. This creates temporary vectorized <filename>.int.c files that contain AltiVec vector macro references.
2. Compiles the <filename>.int.c files with the AltiVec-aware C compiler, creating vectorized object files (<filename>.o).
3. If appropriate, passes the resulting object files, together with any other input files and options specified on the command line, to the compiler for further processing and linking.

So to sum up, the driver acts just like the compiler – except that VAST is invoked "under the covers" to auto-vectorize code for AltiVec.

User interaction

VAST-C/AltiVec is intended for use primarily as an automatic tool; at a minimum, you need to know only how to invoke it (see section 2). However, because of the complexity of the transformations involved and the unavailability of some important data at compile time, the added optimizations VAST-C/AltiVec performs may not significantly decrease the input program's execution time.

If the execution time has not decreased, use the `-Vmessages` switch to enable the vectorization messages. You can also enable VAST-C/AltiVec's listing and look at the diagnostic messages in this special file. Referring to the relevant sections in this document, you may be able to improve the optimization by switching on or off certain transformations or default assumptions, inserting pragmas, or making minor code modifications. In some cases, you may be rewarded with dramatically improved performance for a relatively small effort.

Performance tips

If you have a choice in programming your application, use the smallest data types you can. For example, `long` operations (32 bit) take twice as long in vector mode as `short` operations (16 bit). Also, `float` operations can be vectorized efficiently but `double` operations are not supported by the vector unit.

To vectorize efficiently, your program should be operating on arrays of data in reasonably straightforward loops. Programs which chase pointers in complicated data structures at the lowest level are generally difficult to vectorize. Most importantly, only array uses where consecutive array elements are accessed (unit stride) can make efficient use of the AltiVec unit. This means, for example, that you want to have the rightmost subscript vectorized on multiple-dimension arrays. Also, you want to organize your data as structure of arrays rather than an array of structures where possible.

If you have simple loops traversing arrays with the small types and these loops are still not being vectorized, then you are probably encountering data dependency problems, where vectorizer cannot vectorize because there may be overlap among the arrays. Usually, there is no actual overlap, and there are many ways you can inform the vectorizer that your loops are safe to vectorize. *At a minimum, you should be familiar with the assertion levels (-L), which are discussed in the next chapter. These can be very important to performance on some codes.*

Optimizations

VAST-C/AltiVec's optimizations include:

- Full analysis of large loop nests, with vectorization at the appropriate level.
- Vectorization of non-aligned arrays.
- Restructuring of loop nests to allow enhanced vector execution.
- Vectorization of loops containing reduction operations (such as global sum functions).
- Vectorization of loops with iteration counts that aren't exact multiples of the hardware vector size.
- Unrolling of vectorized loops to increase compiler-scheduled overlap.

How to use this guide

Section 2 describes how to invoke VAST-C/AltiVec. If you want to use VAST-C/AltiVec as a strictly automatic tool, you can skip the remainder of the guide beyond section 2.

Section 3 discusses communication with VAST-C/AltiVec - VAST's messages, and ways to guide VAST action (user pragmas).

Section 4 has information specific to AltiVec vector code generation, with an example of generated code and a discussion of VAST-C features for AltiVec, and coming enhancements.

Concepts and rules of VAST-C/AltiVec's optimization techniques are discussed in the remaining sections. Examples in these sections illustrate optimizable and unoptimizable loops. Of special importance are the sections on data dependency analysis (Section 7) and function inlining (Section 9).

Optional features

Most VAST-C/AltiVec features are turned on by default, but some are not. To enable automatic inline expansion of calls to small functions, use `-Wv, -J1` on the invocation. (See the chapter on Inlining for further details.)

To specify an assertion level, use the `-L` switch. For almost all codes you can specify the switch `-Wv, -L3` and the code will get significantly more vectorization (and more performance).

To see vectorization messages, use the `-vmessages` switch. These messages are not shown by default (as the output can be voluminous) but the messages can be very useful in helping you understand what the vectorizer has discovered in your program.

2. Invoking VAST-C/AltiVec

vcc driver

Normally, VAST-C/AltiVec is invoked the vcc driver. It is used just like your normal compiler, and you can use all your normal compiler switches with it. The driver takes care of vectorizing the code in the VAST-C precompilation phase, and then compiles this vectorized code with the normal compiler. For example,

```
vcc myprog.c
```

Will create an a.out that is vectorized and ready to run on an AltiVec system.

Usage

```
vcc    [<vcc_option>]... [<cc_option>]...  
      [-Wv,<v_option>[,<v_option>]...] [files]
```

where:

<vcc_option>	Represents any vcc driver option (see “vcc Options” below).
<cc_option>	Represents any C compiler option. All normal compiler options can be used.
<v_option>	Represents any VASToption. vcc options and input files may appear in any order.

File extensions

1. filename with a .c suffix: C source file.
2. filename with a .cpp suffix: C++ source file.

3. filename with a .i suffix: preprocessed C source file.
4. filename with a .s suffix: assembler source file.
5. filename with a .o suffix: object file.
6. filename with a .a suffix: archive file.

vcc options

- c** suppress the link editing phase of the compilation and do not remove any object files produced.
- dryrun** display but do not execute cc internal commands. vcc will still check for the existence of essential executable files.
- keep** keep intermediate VAST files. These files are preprocessed C source files. The intermediate files produced are as follows. For an input file "file1.c" the intermediate file will be named "file1.int.c".
- l<suffix>** search the library file lib<suffix>.a.
- L<dir>** search directory <dir> for libraries prior to searching the default directories.
- o<name>** name the final output file <name>. When used with -c, names the object file, otherwise names the link edited output file. (The default link edited output file is named a.out.) -o is ignored if the -vo option is used.
- v** Display vcc internal commands as execution progresses.
- vo** Execute VAST pass only. Certain compiler options included on the command line may be ignored.
- vn** Execute compiler only. vastcav options included on the command line are ignored. Note that the -vo and -vn options are mutually exclusive.
- w** Suppress warning messages.
- Wv** Hand off arguments to VAST. VAST options must be separated by commas and may not contain embedded blanks.
Examples:
-Wv, -Valigned
-Wv, -J1, Vnopinteroverlap
- Yvc, <path>** substitute for cc an alternate executable whose pathname is specified by <path>.
- Yvv, <path>** substitute for vastcav an alternate executable whose pathname is specified by <path>.

Example

```
vcc -O -o file1.vector.exe -Wv,-L3,-Vmessages file1.c
```

Executes `vastcav` with assertion level 3 (-L3), and asks for vectorization messages to be displayed (-Vmessages). Compiles (with compiler optimization -O) the intermediate file using the AltiVec-enhanced target compiler and invokes the linker to produce the executable output file `file1.vector.exe`.

Assertion levels

Level	Description
0	Assume Nothing. (Default)
1	Arguments to functions do not overlap.
2	Pointers do not overlap with non-pointers.
3	Pointers with different names do not overlap.
4	All overlap is explicit.

The table above shows data dependency assertion levels that affect the optimization and vectorization of the input program; these are controlled with -L. Use of the -L switch is heavily encouraged! Use the highest level that you know will work for your program – you will be rewarded with higher performance. In real programs, it is very rare for objects with different names to overlap; a little assistance from you in the form of a -L option can go a long way in improving the optimization of your program.

Higher assertion levels include all lower levels. As an example, -L3 causes VAST-C to assume that pointers with different names do not overlap, pointers do not overlap with non-pointers, and arguments to functions do not overlap.

Notes on assertion levels

Level 0: Assume nothing. This is the default. At this level VAST-C must assume that any pointer may overlap with any other variable, and that arguments to a function may also overlap. This assumption hinders the ability of VAST-C to determine whether or not a data dependency exists in the presence of pointer variables and function arguments. If a data dependency may exist, VAST-C cannot vectorize the loop.

```
func ( a, b )
int *a, *b;
{
    for ( i = 0; i < n; i++ )
        *a++ = *b++;
}
```

Above, VAST-C must assume that pointers `a` and `b` may overlap in memory.

Level 1. -- Assume that any pointer may overlap with another pointer or nonpointer, except that arguments to a function will not overlap with each other. VAST-C will assume, in the above example, that `a` and `b` do not overlap in memory.

Level 2. -- In addition to level 0 and 1 assumptions, also assume that pointers will not overlap with non-pointers.

```
for ( i = 0; i < n; i++ )
    *p++ = a[i];
```

Above, VAST-C can assume that pointer `p` does not overlap with array `a`.

Level 3. -- In addition to level 0, 1, and 2 assumptions, assume that pointers with different names do not overlap.

```
for ( i = 0; i < n; i++ )
    *p++ = *q++;
```

Above, VAST-C may assume that pointer `p` does not overlap with pointer `q`.

Level 4. -- In addition to level 0, 1, 2 and 3 assumptions, assume that no overlap exists unless it is explicit. For example, `*p` may not overlap with `*(p+n)`.

VAST-C/AltiVec switches

To use the VAST-C/AltiVec switches, specify `-Vswitch`. No space is allowed between `-V` and the switch. You cannot use commas to separate switches. For example, these uses are illegal:

```
-V novector                ILLEGAL, space between -V and switch.
-Vleastrecent, cachestrip  ILLEGAL, only one switch per -V.
```

Example with "vcc" driver:

```
vcc -Wv, "-Valigned, -Vnopointeroverlap" -O3 myfile.c
```

These are the switches currently defined for AltiVec:

-Valigned

Asserts that start of all arrays is aligned on 16-byte boundary. This can improve performance by up to a factor of two, if you have arranged to have all of your arrays aligned. The average performance gain is about 20% with this option.

-Vcodespace

Request optimization of code space over time. Turns off vector loop unrolling, for example. May seriously degrade performance.

-Vinline

Request automatic inlining of small "leaf" procedures. May improve performance.

-Vleastrecent

Use least-recently-used version of load/store (so that the vector data does not stay in cache). Default is to use normal (caching) versions. We think that this option will only pay off if you have huge data sets.

-Vmessages

Request VAST to display vectorization messages while processing the input program. These messages are sent to the standard output.

-Vnoargumentoverlap

Assert that arguments to functions don't overlap. It is very important to make this assertion, if you can. This is equivalent to assertion level -L1.

-Vnoassoc

Demand that optimizations that could change the last few bits of the result (relative to the scalar original results) not be done. (No associative transformations). Can seriously degrade performance.

-Vnodepchk

Assume potential dependencies are not real dependencies. May improve performance by allowing more loops to vectorize.

-Vnopointeroverlap

Assert that pointers in vectorizable loops don't overlap. It is very helpful to make this assertion, if you can. This is equivalent to assertion level -L3.

-Vnovector

Turns off vectorization. You can turn it back on for selected loops with the "#pragma vd_vector" pragma. This is useful for selective vectorization.

-Vskip

Skip all transformations. There are some non-vector optimizations that VAST does, this should turn off everything. You can turn on specific loops with the "#pragma vd_vector" pragma.

VAST-C/Altivec options

The options can have various parameters, including:

routines = names of functions, separated by commas.

filenames = Unix file names, separated by commas.

nnn =integer constant

The available options are:

[-I *routines*]

Names of functions that should be expanded inline.

[-J *nnn*]

Level to automatically expand from bottom of call tree (default 1).

[-M *mm*]

Maximum threshold (code lines) for automatic inlining.

[-N *routines*]

List of routines not to inline.

[-S *filenames*]

Files in which to search for routines to be inlined.

[-Y *routines*]

Names of routines that should be expanded inline (including called routines; nested expansion).

3. User pragmas and messages

Overview

You may sometimes have information about the structure of a program and about its data that is unavailable to VAST-C through inspection of individual program units. For this reason, a way for you to guide VAST-C is supplied via user pragmas (also called user directives). User pragmas are passed via the `#pragma` statement provided in ANSI C.

Pragma format

VAST-C pragmas have the format:

```
#pragma vd_ directive
#pragma vd_l directive
#pragma vd_r directive
#pragma vd_f directive
```

The `#pragma vd_` flags this as a pragma to be used by VAST_C. Following is an optional scope parameter. `f` stands for "file" (meaning the directive applies until the end of the input file), `r` stands for "routine" (directive applies until the end of the current function), and `l` for "loop" (directive applies to the next loop encountered). A blank is equivalent to `l`. Some directives ignore the scope parameter.

The body of the directive begins after one or more blanks. Many directives can be preceded by `no`, thus effecting the reverse operation. Directives can be either upper or lower case.

Examples of pragmas:

`#pragma vd_ nodepchk` (*Ignore potential data dependencies in the following loop.*)

`#pragma vd_r novector` (*Turn off vectorization for the rest of this function.*)

`#pragma vd_f noassoc` (*Do no associative transformations for the rest of the file.*)

VAST pragma summary

The full set of pragmas is summarized in the table below. The "scope" entry is either I for "immediate," meaning that the pragma applies immediately; L, meaning that it applies to the next loop; R, meaning that it applies to the whole routine; or **lrf**, which means that any of the loop, routine, or file options can be used to control the scope.

A short description of each of these pragmas follows the table. In addition, the more important pragmas are discussed in detail at the appropriate points in the sections on optimization.

VAST-C/AltiVec pragmas

Pragma	Function	Default	Scope
aligned	Specify that arrays are aligned	n/a	R
skip/noskip	Disable/re-enable transformations	noskip	lrf
nodepchk/ depchk	Do/don't ignore potential overlap of array sections.	depchk	lrf
novector/ vector	Disable/enable vectorization.	vector	lrf
noassoc/ assoc	Don't/do perform associative transformations.	assoc	lrf
permutation	Pass list of integer arrays that have no repeated values.	n/a	R
relation	Specify relationship between two simple variables.	n/a	R
count	Supply iteration count for loop.	n/a	I
iterations	Supply iteration count for classes of loops.	n/a	lrf
nounroll/ unroll	Unroll loop.	unroll	lrf
autoexpand/ noautoexpand	Automatically expand small routines inline.	noauto	R
expand	Expand particular routine(s).	n/a	I
nexpand	Nested expansion of particular routine(s).	n/a	I

Transformation pragmas

These pragmas are used to change the way VAST-C transforms a loop.

aligned

aligned declares arrays to be aligned on a 16-byte boundary. This allows VAST-C/AltiVec to generate code that directly fetches them into vector registers, rather than generating the default code to deal with non-aligned arrays by shifting the values. *Only use `aligned` if you know that the arrays are aligned correctly, otherwise you can get wrong answers.*

novector/vector

novector disables conversion of loops to vector form. `vector` serves only to toggle back from `novector`; it does not force conversion (see `select`). The `-dv` switch is equivalent to `novector` with file scope. `novector` is a subset of `skip`.

skip

skip causes VAST-C to avoid transformation for the directed loop or routine. This is the pragma to use if you want VAST-C/AltiVec to leave a loop unoptimized (no optimizations of any kind). `novector` is a subset of `skip`.

noassoc/assoc

By default, VAST-C transforms certain constructs into optimized or concurrent versions in which the order of operations may be different than the original (they have been associatively transformed). Because of the way numbers are internally represented in computers, this operation reordering may result in answers that differ slightly from the scalar original.

The **noassoc** pragma disables all associative transformations, including generating reductions (such as sum or dot product of arrays), and operation reordering when minimizing dependent regions.

The `-da` switch is equivalent to `noassoc` with file scope.

select

Use this pragma to select a particular loop in a loop nest for vectorization. (This overrides VAST's default selection/threshold criteria relating to percent dependent and percent conditional, and can be considered as a way to "force" vectorization when possible.)

Data dependency pragmas

These pragmas are used to help VAST-C decide whether data dependency conflicts actually exist in a loop. If you know that an operation is not recursive, you can supply one of these pragmas to inform VAST-C. (Data dependency pragmas are discussed further in a later chapter.)

nodepchk/depchk

When elements of an array are modified within a loop, VAST-C must determine the exact storage relationship of these elements to all other references to the array in the loop. This must be done to ensure that the references do not overlap, and thus can safely be executed in vector mode. When the relationships cannot be determined, VAST issues a potential dependency diagnostic. The **nodepchk** pragma asserts that all such potentially recursive relationships are in fact not recursive. It does not, however, force the optimization of operations that are unambiguously recursive. The **depchk** pragma is used only to toggle back to the default state.

permutation

The **permutation** pragma declares an integer array to have no repeated values. This is useful when the integer array is used as a subscript for another array (indirect addressing). If it is known that the integer array is used merely to permute the elements of the subscripted array, then it can often be determined that no feedback exists with that array reference.

relation

The **relation** pragma advises VAST-C that a specified relationship exists between two integer variables or between an integer variable and an integer constant. This information may be useful to VAST-C in resolving otherwise ambiguous array relationships.

DISJOINT

The **disjoint** pragma is used to tell VAST-C that pointers do not overlap. This can be very useful for codes that make heavy use of pointers. (The C99 keyword **restrict** is even better.)

Vectorization messages

You can get vectorization information from VAST by turning on the vectorization messages with the `-Vmessages` switch, for example:

```
vcc -Wv,-Vmessages myfile.c
```

This will send vectorization messages to the standard output.

Diagnostic messages

Each vectorization message includes the line number and (if relevant) a variable name. These messages are VAST's means of notifying you of what problems it encountered in optimizing the loops in the source program. There are several different types of diagnostics.

Optimization inhibition messages

Translation Diagnostic. Describes a problem or potential problem in executing a loop in vector mode. May prevent loop from being optimized.

Data Dependency Conflict. A real or potential feedback from one loop pass to the next prevents the safe use of parallel operations. Potential feedback may result in generation of alternate versions of the loop. Otherwise feedback may prevent at least part of a loop from being optimized. (Consider using a `-L` assertion level or a `NODEPCHK` pragma if you know the loop is actually safe to vectorize.)

Informative messages

Warning Message. Some potentially troublesome input has been encountered.

Note Message. Tells about some opportunity or action on the input that might be of interest.

Error messages

Syntax Error. A construct that is not legal in C has been encountered. No translation is done for this program unit.

Internal Error. An internal problem with VAST-C has been detected. No further processing is done for this subprogram; translation is attempted again for the next subprogram in the input file. Please report the error.

You can suppress each of these types of messages independently via the `-q` parameter on the VAST-C command line or on the `SWITCH pragma` (see section 3).

4. AltiVec Code Generation

Overview

VAST-C/AltiVec has several features to take advantage of the capabilities of the AltiVec. These include:

- Dealing with arrays that are not aligned on 16 byte boundaries.
- Dealing with iteration counts that are not an even multiple of the vector register size.
- Vectorizing loops with array reduction operations.
- Unrolling the vector strip loop.
- Handling vectors that have non-unit stride.
- Vectorizing loops that contain indirect vector addressing (gather/scatter).
- Optimization of complex vectors, when the complex values are stored as two-element structures.

AltiVec code generation overview

Normally, you do not need to examine the vector code generated by VAST-C/AltiVec. This code, while generated at the C level, actually maps almost one-for-one with the vector instructions of the AltiVec unit; thus, it is fairly low-level and can appear voluminous in the output.

If you have some familiarity with the AltiVec unit, you may wish to examine the VAST-C/AltiVec code. You can sometimes shuffle the generated vector instructions to achieve slightly better performance - you would do this by editing the VASTed output. This is recommended for expert users only -- most users will not want to do this. To get the VASTed output, use the `-keep` switch

on the vcc driver command. This will keep the VAST intermediate file around (these files have the compound extension `.int.c.`)

Vector code generation sections

If you examine the code generated by VAST-C/AltiVec for a vectorized loop, you will find the following basic structure:

1. Run time test to see if loop should be vectorized. This may be a test to see if the iteration count is greater than zero, for example.
2. Vector declarations.
3. Initializations.
4. Vector strip loop.
5. Main vector calculation.
6. End of vector strip loop.
7. Cleanup for remaining iterations and for non-aligned vectors.
8. End of vector code.

Vector code generation example

Here is an example of VAST/AltiVec code generation (actual code generated may differ in more recent VAST versions):

```
void daxpy(n,da,dx,incx,dy,incy)
float dx[],dy[],da;
int incx,incy,n;
{
    int i,ix,iy,m,mp1;
#pragma vd_ nodepchk
    for (i = 0;i < n; i++)
        dy[i] = dy[i] + da*dx[i];
}
```

And here is the vectorization:

```
/* Translated by Pacific-Sierra Research VAST-C AltiVec 5.2 B 14:52:14 8/13/99 */
void daxpy( n, da, dx, incx, dy, incy )
float dx[], dy[], da;
int incx, incy, n;
{
    int j1, j2, j3, j4, j5, j6, j7, j8, j9, j10, j11, j12, j13, j14, j15,
    j16;
    int i, ix, iy, m, mp1;
    {
        int j4s;
        if ( n > 0 )
        {
```

```

vector float r2v; // Declarations
vector float dy1v, dx1v, r1v;
vector float dy18v, dy19v, dx9v, dx10v, dy20v, dy21v;
vector float dy14v, dy15v, dx7v, dx8v, dy16v, dy17v;
vector float dy10v, dy11v, dx5v, dx6v, dy12v, dy13v;
vector float dy2v, dy3v;
vector unsigned char dy4v = vec_lvsl(0, &dy[0]);
vector float dx2v, dx3v;
vector unsigned char dx4v = vec_lvsl(0, &dx[0]);
vector float dy5v, dy6v, dy7v;
vector unsigned char dy8v = vec_lvsl(0, &dy[0]);
vector unsigned long dy9v = vec_perm(vec_splat_u32(0),
vec_splat_u32((-1)), dy8v);
*(float *)&r2v = da; // Initializations
r1v = vec_splat(r2v, 0);
j1 = 0;
j7 = n;
dy2v = vec_ldl(j1 * sizeof(float ), &dy[0]);
dx2v = vec_ldl(j1 * sizeof(float ), &dx[0]);
dy5v = dy2v;
for ( j2 = 0; j2 < (j7 - 4 * 4) + 1; j2 += 4 * 4 ) // Vector Strip Loop
{
    j4 = (j1 + j2) * sizeof(float );
    j3 = j4 + 4 * sizeof(float );
    j11 = j3 + 4 * sizeof(float );
    j12 = j4 + 4 * sizeof(float );
    j13 = j3 + (4 * sizeof(float )) * 2;
    j14 = j4 + (4 * sizeof(float )) * 2;
    j15 = j3 + (4 * sizeof(float )) * 3;
    j16 = j4 + (4 * sizeof(float )) * 3;
    dy3v = vec_ldl(j3, &dy[0]);
    dy1v = vec_perm(dy2v, dy3v, dy4v);
    dx3v = vec_ldl(j3, &dx[0]);
    dx1v = vec_perm(dx2v, dx3v, dx4v);
    dy1v = vec_madd(r1v, dx1v, dy1v);
    dy6v = vec_perm(dy1v, dy1v, dy8v);
    dy7v = vec_sel(dy5v, dy6v, dy9v);
    vec_stl(dy7v, j4, &dy[0]);
    dy10v = vec_ldl(j11, &dy[0]);
    dy11v = vec_perm(dy3v, dy10v, dy4v);
    dx5v = vec_ldl(j11, &dx[0]);
    dx6v = vec_perm(dx3v, dx5v, dx4v);
    dy11v = vec_madd(r1v, dx6v, dy11v);
    dy12v = vec_perm(dy11v, dy11v, dy8v);
    dy13v = vec_sel(dy6v, dy12v, dy9v);
    vec_stl(dy13v, j12, &dy[0]);
    dy14v = vec_ldl(j13, &dy[0]);
    dy15v = vec_perm(dy10v, dy14v, dy4v);
    dx7v = vec_ldl(j13, &dx[0]);
    dx8v = vec_perm(dx5v, dx7v, dx4v);
    dy15v = vec_madd(r1v, dx8v, dy15v);
    dy16v = vec_perm(dy15v, dy15v, dy8v);
    dy17v = vec_sel(dy12v, dy16v, dy9v);
    vec_stl(dy17v, j14, &dy[0]);
    dy2v = vec_ldl(j15, &dy[0]);
    dy19v = vec_perm(dy14v, dy2v, dy4v);
    dx2v = vec_ldl(j15, &dx[0]);
    dx10v = vec_perm(dx7v, dx2v, dx4v);
    dy19v = vec_madd(r1v, dx10v, dy19v);
    dy5v = vec_perm(dy19v, dy19v, dy8v);
    dy21v = vec_sel(dy16v, dy5v, dy9v);
    vec_stl(dy21v, j16, &dy[0]);
}
if ( (j7 & 4 * 4 - 1) == 0 )
    dy6v = dy5v;
else {
    for ( ; j2 < (j7 - 4) + 1; j2 += 4 ) // Unrolling Cleanup
    {
        j4 = (j1 + j2) * sizeof(float );
        j3 = j4 + 4 * sizeof(float );
        dy3v = vec_ldl(j3, &dy[0]);
    }
}

```

Iteration count cleanup

VAST generates special code to compute the remaining loop iterations, when the iterations desired are not a multiple of the vector size. The final elements are stored element-by-element so that only the appropriate elements in the result arrays are changed.

Array reductions

Reduction operations include the summation of all elements of an array, finding the maximum element in an array, and taking the dot product of two arrays. Reduction operations take array-valued data and reduce it to a scalar value.

Processing reductions takes special code on the AltiVec to create a vector of “partial reductions” that is then reduced into the resulting scalar.

Example:

```
for (i=0;i<n;i++)    (will vectorize)
    s = s + a[i]*b[i]
```

Vector strip unrolling

VAST unrolls the vector strip loop to provide more instructions to overlap for the AltiVec compiler. This can provide substantially improved performance. VAST will automatically unroll the vector strip loop where it may provide a performance benefit, and will unroll it either two or four times.

Loops with four or less unique vector references will be unrolled four times, loops with 24 or less unique vector references will be unrolled twice. Otherwise, the loop is not unrolled.

In the example on the previous pages, the vector strip loop was automatically unrolled four times.

Non-contiguous vectors

While the AltiVec instructions operate on contiguous 16-byte entities, VAST can vectorize loops that contain strided vectors or gather and scatter operations. VAST will create scalar instructions to explicitly move the data into contiguous areas for subsequent vector operation.

Example:

```
for (i=0;i<n;i++)    (will vectorize)
    a[i] = a[i]/b[i*2] + sqrt(c[ic[i]]) -x
```


5. Optimization overview

This section presents some general features of VAST-C's optimization.

Pointers and array references

A pointer is a variable that contains the address of another variable. The use of pointers can lead to more compact and efficient coding, but also can create programs which are more difficult to analyze. The use of pointers makes it especially difficult to determine whether or not data dependencies exist. For the most part, pointers are equivalent to array indexing and, viewed as such, the existence of overlap can be determined.

The availability of pointers in C gives the programmer a great deal of flexibility. Arrays of data can be referenced in many different ways. VAST-C recognizes and understands these different uses. Below are three equivalent examples, all of which VAST-C optimizes.

```
float *a, *b, x;

/* loop 1 */
for ( i = 0; i < n; i++ )
    *(a+i) = *(b+i) + x;

/* loop 2 */
for ( i = 0; i < n; i++ )
    a[i] = b[i] + x;

/* loop 3 */
i = 0;
while ( i++ < n )
```

```
*a++ = *b++ + x;
```

Figure 6.1 Equivalent loops.

Pointers can be vectorized in the above examples (in the same manner as arrays) if we can determine that a and b do not overlap. This is done either through an invocation option to VAST-C, a user `pragma`, program analysis by VAST-C, or a runtime test generated by VAST-C. A full discussion on pointers and their effect on data dependency analysis can be found in the "Data Dependency Analysis" section of this guide.

Understanding loop variables

Understanding loop optimization is made possible by understanding the optimizable and non-optimizable uses of the variables in a loop. Every variable in an optimizable loop can be categorized as one of three things: scalar, index, or vector. A scalar is a single value which does not change through all the iterations of the loop. An index is an integer quantity which is incremented by a constant amount each pass through the loop. A vector is a range of memory locations, with a constant skip or stride between consecutive elements.

Here is an example of each of these:

Loop:

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

Classification:

i,j: index variables

a,b: vector

x : scalar

Later sections in this document examine each of these kinds of variables in detail, and explore how their different uses can make a loop optimizable or not.

Loop types

for, while, and do loops are considered for optimization.

"for" loops

VAST-C analyzes entire nests of for loops (including converted if loops) for possible optimization.

```
for ( i = 0; i < n; i ++ )  
    for ( j = 0; j < m; j++ )
```

```
a[i][j] = b[i][j] * c[i][j];
```

"while" loops

VAST-C analyzes while loops. while loops must have a fixed iteration count to qualify for optimization.

```
while ( i < n ) {  
    *a++ = *b++ ;  
    i++;  
}
```

(The iteration count is n-i.)

"do" loops

Similar to while loops, VAST-C analyzes do loops. Again the loop must have a fixed iteration count to qualify for optimization.

```
do {  
    *a++ = *b++;  
    i++;  
} while ( i < n );
```

(The iteration count is max(1,n-i).)

Allowed statements

Statements that may appear in an optimizable loop are listed below.

Expressions

```
a[i] = b[i] + c[i];
```

Compound statements

```
{ ... }
```

Selection statements

```
if, else, switch, case, default
```

Iteration

```
while, do...while, for
```

Jump

```
goto, continue, break
```

Iteration count

For many of its optimizations, it helps VAST-C to have some idea of the iteration counts of loops in the program. Very short loops are often better off translated in a different way than loops with many iterations. In choosing vector loops, knowing approximate values for the iteration counts is extremely useful.

If the iteration count of a loop is constant, VAST-C understands it already. In cases where the iteration count is a variable, VAST-C looks at the declared dimensions of arrays used in the loop in order to determine what the maximum iteration count could be.

If the iteration count is variable and cannot be determined from the information in the routine until execution time, but you know the approximate number of iterations, you can use the `count` or `iterations` pragma to provide this information to VAST-C. The `count` pragma has the format:

```
#pragma vd_ count ( n1 )
```

The `iterations` pragma has the format:

```
#pragma vd_ iterations(var1=n1[,var2=n2,...])
```

where

`n1, n2...`

The assumed iteration count values. These do not have to be exact as they are only used as guidelines by VAST-C.

`var1, var2...`

The indices of a loop with the given assumed iteration count values. The `count` pragma can be used at the file (`f`), routine (`r`), or at the local (`l`) level. The `iterations` pragma may only be used at the routine level. A `vd_f count(0)` or a `noiterations` pragma returns VAST-C to its normal iteration count processing.

```
optim6 ( a, b, n )
float *a, *b;
int n;
{
  int i;
#pragma vd_ count (3)
  for (i=0; i<n; i++)    (Not optimized, iteration count is too small.)
    *(a+i) = *(b+i);
}
```

Below is an example with a small iteration count on the inner (fast) dimension.

```
optim7 ( a, b, c, l, m )
float a[100][3], b[100][3], c[100][3];
int l, m;
{
```

```

int i, j;

#pragma vd_ iterations ( j=3, i=100 )
for ( i=0; i < m; i++ )
    for ( j=0; j < l; j++ )
        a[i][j] = b[i][j] + c[i][j];

```

Structures

VAST-C can optimize loops that contain structure references. Optimization can occur both across structures and within structures, at the same time. A loop across structures can be optimized.

```

typedef struct tag { int xaxis[1000],
                    yaxis[1000]; } Cartesian;

Cartesian box1, *pbox1;
int slope[1000];
pbox1 = &box1;
for ( i = 0; i < 1000; i++ )
    slope[i] = ( pbox1->yaxis[i+1] - box1.yaxis[i] ) /
               ( pbox1->xaxis[i+1] - box1.xaxis[i] );

```

The above example shows the use of elements within a structure, including the use of structure pointers. Below is an example of a loop using elements across structures.

```

struct grade { float math, english,
              history; } students[100];

math_average = 0;
for ( i = 0; i < 100; i++ )
    math_average += students[i].math;
math_average /= 100.0;

```

Assignment operators

VAST-C fully understands the use of all assignment operators, multiple assignments, and nested assignments and will optimize loops which contain such constructs.

```

for ( i =0; i < 100; i++ {
    a[i] = b[i] = (float) 100;
    c[i] = ( b[i] = b[i] + 3 ) + a[i];
    d[i] *= c[i];
}

```

```
}
```

Translation diagnostics on loops

Some of the more common translation diagnostics that may be issued for loops are shown in this section.

unable to get constant length from loop

```
while ( *a > 0 )
    *b++ = sqrt(*a++) ;
```

VAST-C issues this diagnostic when it cannot convert a `for`, `while`, or `do` loop to a form that has a fixed iteration count known before the loop is executed.

user function references not allowed in iteration count

```
for ( i=0; i < nlen(j,k); i++ )
    a[i] = 0.
```

In this example `nlen` is an external user function. Such functions cannot appear in the iteration count expression for a `for` loop (they cannot be in the start, end or increment fields of the `for` statement). Statement functions are allowed, however.

iteration count too short

```
for ( i=0; i < 2; i++ )
    a[i] = b[i];
```

When loops have an explicit iteration count which is less than a set cut-off amount they are not vectorized as they will probably run faster in scalar mode.

array dimensions indicate iteration count is too short

```
float a[2], b[2];
for ( i=0; i < n; i++ )
    a[i] = b[i];
```

The declared size of an array dimension may limit the maximum possible iteration count to less than the cut-off (see above).

6. Loop Optimizations

Overview

This section describes some of the features that VAST-C provides for loop optimization. As most of the time is spent in loops, optimizing them can be very important to final performance.

The transformations in this sections are shown in standard C rather than the actual final AltiVec code, as the AltiVec code can be large and might obscure the transformation.

Loop fusion

VAST-C combines consecutive loops that have no statements between them, have the same iteration count, and will give the same answers when merged. This optimization helps expose common subexpressions and reduces overhead. There is a default maximum of five loops and 50 total lines of fused code, as overly large loops might cause the compiler to run out of registers.

Below is an example of two loops where common subexpression analysis will eliminate repeated operations. `c[i]` will need to be loaded only once and `c[i]*.5` will need to be calculated only once.

```
for ( i=0; i < 100; i++ )
    a[i] = b[i] + c[i]*.5;
for ( i=0; i < 100; i++ )
    e[i] = d[i] + c[i]*.5;
```

Below is the fused form of the loops.

```
for ( i=0; i < 100; i++ ) {
```

```

    a[i] = b[i] + c[i]*.5;
    e[i] = d[i] + c[i]*.5;
}

```

Loop re-rolling

Optimizing compilers for some high-speed scalar computers shuffle the order of the instructions they produce in order to overlap operations as much as possible. A common technique used to aid such compilers is to "unroll" short loops to give the compiler more flexibility in reordering instructions within a single loop pass. For example, a simple dot product

```

for ( i=0; i < n; i++ )
    s += a[i];

```

might be unrolled like this:

```

for ( i=0; i < n-3; i +=4 )
    s += a[i] + a[i+1] + a[i+2] + a[i+3];

```

The second loop performs the same function as the first, but it does four iterations per pass instead of one. The unrolling technique is generally undesirable on vector machines, since it decreases vector length and increases indexing overhead. In some cases, VAST-C can detect unrolled loops and "re-roll" them into their original form.

A re-rollable loop must have an explicit constant non-unit increment specified on the for statement. It must contain no data dependencies. An unrolled summation or dot product must add each operand to the reduction scalar in index order. The number of assignments in an unrolled assignment must be the same as the increment on the for loop, and each assignment must do the next computation in index order. Below is an example of a loop that VAST-C can re-roll.

```

for ( i = 0; i < 997; i +=3 ) { (Rerolled into one vector add.)
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}

```

Translation:

```

for ( i = 0; i < 1000; i++ )
    a[i] = b[i] + c[i];

```

Loop unrolling

Loop unrolling is of benefit for loops whose scalar optimization is inhibited because they have too few operations per pass. Loop unrolling reduces the

percentage of time spent in loop overhead, and provides more instructions for the compiler to overlap in each pass of the loop.

VAST-C can unroll automatically or a pragma may be used to force it to unroll. Automatic unrolling is done when VAST-C finds a case it believes will benefit from this treatment; you may request unrolling for other loops with the unroll pragma.

```
#pragma unroll
for ( i = 0; i < n; i++ )
    d[i+1] = a[i] + b[i]*c[i];
```

Translation:

```
j1s = n % 4;                                (Unrolled 4 times.)

for ( i=0; i < j1s; i++ )                  (Cleanup loop)
    d[i+1] = a[i] + b[i]*c[i];

for ( i=j1s; i < n; i+=4 ) (If N<4, this loop not done.)
{
    d[i+1] = a[i] + b[i]*c[i];
    d[i+2] = a[i+1] + b[i+1]*c[i+1];
    d[i+3] = a[i+2] + b[i+2]*c[i+2];
    d[i+4] = a[i+3] + b[i+3]*c[i+3];
}
```

Also, when the loop iteration count is 4 or less and the body of the loop is small, the loop will be completely unrolled and replaced with assignment statements.

```
for (j=0; j<m; j++) {
    for(i=0; i<3; i++) *(a+i+j*100) = 0.0;
}
```

Translation:

```
for ( j=0; j < m; j++ )
{
    a[j*100] = 0.0;
    a[1+j*100] = 0.0;
    a[2+j*100] = 0.0;
}
```

To allow you control over the unrolling process, VAST-C provides the unroll pragma, as follows:

```
#pragma vd_ {l,r,f} unroll [(number_of_times)]
```

where number_of_times is the number of times to unroll the body of the loop (the number of copies of the statements in the loop that appear in the new,

unrolled, loop.) If no argument is supplied, VAST-C calculates the unrolling depth based on default parameters. A value of zero as the `number_of_times` field turns off unrolling. The `nounroll` pragma can also be used to turned off unrolling.

Explicit unrolling is applied at an early stage in VAST-C and thus has relatively few restrictions.

Loops are automatically unrolled whenever certain system-specific conditions are met. For AltiVec code generation, the vector strip loop is frequently unrolled (see previous section).

"while" to "for"

while loops are converted to iterative for loops when a fixed iteration count can be extracted from the loop. This may allow further optimization.

```
while(i > 0) {
    *(a+i-1) = 0.0;
    i -= 1;
}
```

Translation:

```
ilx = i;
if( ilx > 0 )
{
    i2x = ilx > 0 ? ilx & 1 : 0;
    if( i2x == 1 )
        a[ilx-1-1] = 0.0;
    for( i=i2x + 1; i <= ilx; i += 2 )
    {
        a[ilx-1-i] = 0.0;
        a[ilx-2-i] = 0.0;
    }
}
```

Common Subexpression Enhancement

Common expressions within a loop can be collected and computed only once, even when the expression must be permuted to expose the commonality.

As the answers may be slightly different due to the finite word size, this optimization is only done when associative transformations are allowed. VAST-C rewrites the expression with additional parentheses to make the common expression apparent to the compiler.

```

for ( i=0; i<n; i++) {
    *(a+i) = *(b+i) * *(c+i) * *(d+i);
    *(e+i) = *(c+i) * *(d+i) * *(f+i);
}

```

Translation:

```

for ( i=0; i < n; i++ ) {
    a[i] = (c[i] * d[i]) * b[i];
    e[i] = (c[i] * d[i]) * f[i];
}

```

Scalar division removal

VAST-C converts division by scalar to multiplication by the reciprocal, with the reciprocal calculated outside the loop. This transformation will only be done when associative transformations are allowed by the user, as the answers may be slightly different.

This transformation is done only for floating point division.

```

for(i=0; i<n; i++) *(a+i) = x + *(b+i)/s;

```

Translation:

```

float r1s;
r1s = 1. / s;
for ( i=0; i<n; i++ )
    a[i] = x + b[i] * r1s;

```

Loop interchange

For the AltiVec unit, vectors with a stride of one can be processed much more efficiently. In the example below, the loops are interchanged to move the loop that has unit stride and a larger iteration count to the inside.

```

for (i=0; i<100; i++)
    for (j=0; j<10; j++)
        a[j][i] = b[j][i];

```

Translation:

```

for ( j=0; j < 10; j++ )    (now the outer loop)
    for ( i=0; i < 100; i++ )    (now the inner loop)
        a[j][i] = b[j][i];

```

Loop nest collapse

Other things being equal, long vectors are more efficient than short vectors. A common technique for increasing average vector length relies on the conventional storage pattern of C arrays. Using this technique, the last N dimensions of an array can be treated as a singly-dimensioned array whose length is the product of the sizes of the N dimensions. Thus an array declared `a[5][7]` can be treated as if it had been declared `a[1][5*7]`.

Under certain restrictive conditions, VAST-C uses this technique to automatically collapse loop nests into a single loop whose iteration count is the product of the iteration counts of the uncollapsed loops. This transformation can be disabled by the `-d p` option switch.

Collapse criteria:

- The loops must be tightly nested, and there must be one loop index per optimized dimension.
- The loop bounds must be identical to the array bounds.
- There must be no recursion in the loops.
- There must be no use of vector indices outside of array subscripts.
- All vector array references in the loop must conform, that is, the last N subscripts of each reference must be identical to the last N subscripts of all the other vector array references, where N is the nesting depth. Each of the N subscripts must be indexed by one and only one of the loop indexes, with a stride of one. The last N-1 dimensions of the declarations of these arrays must also conform.

```
float a[12][10], b[12][10];
for ( i = 0; i < 12; i++ )
    for ( j = 0; j < 10; j++ )
        a[i][j] = b[i][j] * b[i][j];
```

Here is the collapsed form of the loop:

```
float a[12][10], b[12][10];
for ( j = 0; j < 120; j++ )
    a[0][j] = b[0][j] * b[0][j];
```

Outer loop unrolling

This transformation unrolls outer loops *inside of* inner loops, providing more instructions to optimize in the inner loop without decreasing the iteration count of the inner loop. This optimization is especially useful when it allows common expressions along the outer loop to be more easily eliminated. For example, in

the loop below `c[i]` can be fetched one time and used four times in the unrolled loop.

```
for(j=0; j<100; j++)  
    for(i=0; i<m; i++) a[j][i] = b[j][i] + c[i];
```

Translation:

```
for( j=1; j <= 100; j += 4 ) {  
    for( i=0; i < m; i++ ){  
        a[j-1][i] = b[j-1][i] + c[i];  
        a[j][i] = b[j][i] + c[i];  
        a[1+j][i] = b[1+j][i] + c[i];  
        a[2+j][i] = b[2+j][i] + c[i];  
    }  
}
```

7. Data dependency analysis

Overview

VAST-C/AltiVec ensures that the optimized code gives the same answers as the original. For certain loops, parallel or vector execution would result (or could result) in incorrect answers. A loop in which results from one loop pass feed back into a future pass of the same loop is said to have a *data dependency conflict* and may not be completely optimized. (Such a loop is also said to be recursive or to contain recurrences.) In these cases, VAST-C/AltiVec detects the problem, reports it to the user, and leaves the loop in its original form.

VAST-C/AltiVec does extensive analysis of the arrays used in each loop nest, and examines the offset and stride of accesses to elements along each dimension of arrays that are both used and stored in a loop. If the uses and stores overlap on different iterations of a loop, or if they could possibly overlap, then there is a data dependency problem. In this case the order of execution of the iterations could change the results, and the order of execution of iterations where a loop is parallelized is not known, so the loop cannot be parallelized. Avoiding dependencies is important in getting the highest performance; this section describes some pragmas that VAST-C/AltiVec provides to help you do this.

Data dependency examples

In the loop shown below, the reference to `a[i-2]` at the top of the loop conflicts with the store into `a[i]` at the bottom. VAST-C prints a message to this effect and does not optimize the loop.

```
for ( i=3; i <= 100; i++ )      (Not optimized)
{
    t = a[i-1] + a[i-2];
```

```

    b[i] = t + 3.0 + a[i];
    a[i] = sqrt(b[i]) - 5.0;
}

```

VAST-C uses information from other array dimensions as part of its analysis where possible. The loop in the following example is optimized because VAST-C can see that the second dimension indexes of the a references can never be equal (since n is not equal to $n+1$) and thus there is no recursion (The references to a are to two totally different column vectors - they do not share data).

```

for (i=2; i<= 100; i++ ) (Optimized.)
    a[n][i] = a[n+1][i-1]*b[i] + c[i];

```

Function arguments

In ANSI C it is possible for function arguments to overlap in memory. While this is rarely the case, lacking further information, VAST-C must take the safe road and avoid optimizing loops which involve function arguments appearing on both the left and right sides of an assignment operator.

```

func( pa, pb )
int *pa, *pb;
{
    .
    .
    .
    for ( i = 0; i < 100; i++ )
        *(pa+i) = *(pb+i);
    .
    .
    .
}

```

If, in the above example, pa and pb overlap in memory in a recursive way, then vectorization of the loop would result in incorrect results.

```

int *a;
.
.
func( a, a-1 );

```

As the example above shows, the arguments overlap, so in the function `func` the loop is recursive. As we said, it is very rare for recurrence to exist due to function arguments. For this reason, the assertion level switch is provided to inform VAST-C that arguments never overlap. If you had specified an assertion level of 1 or higher (`-L1`), the loop in `func` could be optimized.

Pointers and data dependency

Pointers are a powerful language feature, and while they can produce more compact and efficient code, they can also present difficulties for data dependency analysis.

```
for ( i = 0; i < n; i++ )
    *pa++ = *pb++ + *pc++;
```

If there is no other information, VAST-C cannot determine whether overlap exists in such cases as the above since `pa`, `pb`, `pc` may overlap. In such a simple case, VAST-C may generate both vector and scalar versions together with a runtime test to determine whether these pointers overlap. As you might suspect, the testing code can mushroom quickly if the number of pointers involved becomes too large. In fact, it would be possible for the if test to outweigh the benefit of the vector code. A simple and efficient way of handling the problem of potential overlap of pointers is provided via pragmas and optimization levels. The `disjoint` pragma provides the user with a way to inform VAST-C of the characteristics of given pointers.

```
#pragma vd_ disjoint ( pa, pb, pc )
for ( i = 0; i < n; i++ )
    *pa++ = *pb++ + *pc++;          (Vectorized.)
```

Using the `disjoint` pragma here informs VAST-C that `pa`, `pb` and `pc` are disjoint -- they do not overlap. Given this information, the loop will be optimized.

A more convenient way for you to help VAST-C with pointers is provided with the assertion level switch. This switch informs VAST-C of the 'style' in which the program has been coded.

Generally, the programmer knows whether or not they have used pointers in an overlapping manner. They may know that only pointers of the same name overlap thereby allowing VAST-C to automatically optimize loops that contain pointers to different names. In the previous example, if you had specified an assertion level of 3 or higher (`-L3`), then VAST-C would know that `pa` does not overlap with any pointers that are not of the same name. Therefore, `pa` does not overlap with `pb` or `pc`.

In some instances, a programmer may use multiple references of the same pointer in a loop, yet recurrences do not exist. For these cases assertion level 4 is provided. At this level only explicitly coded feedback hinders optimization.

```
for ( i = 0; i < n; i++ )
{
    x = *pa++;
    *pa = x;
}
```

It may not be obvious, but feedback is explicitly coded in the above example, and the loop would not be optimized at any assertion level. We can see that the

contents of the location pointed at by `pa` are being stored into the location of `pa+1`. Perhaps more obvious would be the equivalent:

```
for ( i = 0; i < n; i++ )
    *(pa+i+1) = *(pa+i);
```

In addition to the disjoint pragma and the assertion level switch, the `nodepchk` pragma may always be used to inform VAST-C that no data dependency exists.

Potential dependency run-time testing

When it is unclear whether a loop is data-dependent or not, because of variables in array subscripts, VAST-C will generate two versions of the loop together with a run-time test. If the loop is not data-dependent, a vector version of the loop will execute; otherwise the scalar version.

```
dadp02( a, b, ip1, n )
int ip1, n;
float a[], b[];
{
    int i;
    for ( i = 0; i < n; i++ )
        a[ip1+i] = a[i] + b[i]; (Potentially dependent.)
}
```

Translation:

```
if ( abs( ip1 ) >= n || ip1 == 0 )
{
    (vector version)
}
else
{
    for ( i=0; i <= n - 1; i += 1 )
        a[ip1+i] = a[i] + b[i];
}
```

It is possible for array constants to conflict with vector array references. Because of this, the following cannot be safely optimized (`j` may be between 2 and `n`) unless an if statement is placed around the translated code to ensure that this is not the case. VAST-C also handles loops of this kind by generating alternate code.

```
dadp03 ( a, b, n, j )
float *a, *b;
for ( i = 1; i < n; i++ ) (Cannot be optimized unconditionally)
    *(a+i) = *(a+j) - *(b+i);
```

Translation:

```
if ( j < 1 || n - 1 < j )
{
#pragma vdir nodep
    for ( i=0; i < n - 1; i++ )
        a[1+i] = a[j] - b[1+i];
}
else
{
    for ( i=1; i <= n - 1; i += 1 )
        *(a + i) = *(a + j) - *(b + i);
}
```

Ambiguous subscript resolution

When it is not possible with information contained in the loop to determine the relationship between two references to an array, the situation is called ambiguous subscripting or potential dependency. In these situations, VAST-C looks for statements outside of the loop that may provide information which clears up the ambiguity. In the loop below, VAST-C finds the assignment to `n2`, which makes it clear that `n1` is never equal to `n2`, and thus there is no feedback.

```
n2 = n1 + 1;
for ( i = 0; i < n; i++ )          (Optimized.)
    a[n1][i+1] = a[n2][i] * b[i];
```

In the example below, the stores into pointer variables outside the loop are analyzed to determine that there is no dependency.

```
pa = &a;
pb = pa + 100;
for ( i = 0; i < 100; i++ ) (Optimized.)
    *pa++ = *pb++ + x;
```

Data dependency pragmas

There are a number of different pragmas that allow you to help VAST-C optimize more code. These pragmas tell the system information about the variables in the loop, so that better data dependency decisions can be made.

nodepchk pragma

The `nodepchk` pragma asserts that no vector overlap exists in the loop, and thus no synchronization is needed. This is the pragma to use for potentially

vectorizable loops that have hidden relationships between variables. This capability should be used only when you know that no real overlap exists. When it detects potential feedback, VAST-C issues a message asking you to apply this pragma if the loop is in fact not recursive.

Let's look at an example of a situation where you may want to disable data dependency checking. If you know that the variable `k` has a value greater than 999, then there is no overlap between the references to array `a` in the loop below, and you can use the `nodepchk` pragma to tell VAST-C that the loop can be optimized.

```
#pragma vd_ nodepchk
for ( i = 0; i < 1000; i++ )
    a[i+k] = a[i] + b[i] ;
```

disjoint pragma

You can use the `disjoint` pragma to inform VAST-C that certain pointers are disjoint, that is, the memory ranges they point to do not overlap.

Syntax:

```
#pragma vd_ disjoint [(pa,pb,...,pn)]
```

`disjoint` states that the pointers listed do not overlap with each other or any other variables. If no list of pointers is given, all pointers are assumed to be disjoint.

```
#pragma vd_ disjoint ( pa, pb )
for ( i = 0; i < n; i++ )    (Vectorized.)
    *pa++ = *pb++;
```

Since `pa` and `pb` are pointers and may overlap, it would be unsafe to optimize the above loop without the `disjoint` pragma. The `nodepchk` pragma may also be used in this case.

relation -- specifying relations between variables

You can use the `relation` pragma to provide additional information to VAST-C about array subscript ranges, to help in determining if a loop is safe to optimize. The `relation` pragma has the form:

```
#pragma vd_ relation ( simple1 rel simple2 )
```

where `simple1` and `simple2` are simple integer variables (one of them can be an integer constant), and `rel` is one of `>`, `<`, `>=`, `<=`, `==`, `!=`, with the normal C meanings.

When VAST-C cannot otherwise figure out whether the relationship between two uses of an array is recursive, it searches the relations supplied by the user for the current routine to see if they help.

`relation` pragmas are informative only and do not force any action.

`relation` pragmas apply for the whole program unit. If conflicting relations are given, the result is unpredictable. It is up to you to ensure that the relations specified are correct and consistent.

```
#pragma vd_ relation ( j >= n )
...
for ( i = 0; i < n; i++ )    (If j >= n, no overlap.)
    a[i+j] = a[i] + b[i];    (Vectorized.)
```

The `relation` pragma is provided for situations where you are unsure if the `nodepch` pragma (a blanket assertion of non-recursion) is safe, or know it is not, but have some information about relative values of index variables.

permutation -- declaring safe indirect addressing

When an array with a vector-valued subscript appears on both sides of an assignment operator in a loop, feedback is possible even if the subscript is identical. Feedback will occur if there are any repeated elements in the subscripting array.

Sometimes, indirect addressing is used because the elements of interest in an array are sparsely distributed; in this case an integer array is used to point at the elements that are really wanted, and there are no repeated elements in the integer array (e.g., the example below).

This information can be passed to VAST-C through the `permutation` pragma. `permutation` asserts that the named integer arrays contain no repeated elements (i.e., they serve merely to permute the elements of the arrays they indirectly address).

Syntax:

```
#pragma vd_ permutation ( ia1, ia2, ... , ian )
```

`permutation` declares the integer arrays (`ia1`, etc.) to have no repeated values for the entire routine.

```
#pragma vd_ permutation ( ipnt )    (ipnt has no repeated values.)
...
for ( i =0; i < 100; i++ )
    a[ipnt[i]] = a[ipnt[i]] + b[i];
```

Data dependency conflict messages

Data dependency messages are always followed by the name of the variable which is causing the problem and index of the loop causing the problem is given as well.

feedback of array elements

```
for ( i =0; i < 100; i++ )
    a[i+3] = a[i] + b[i];
```

Feedback of results makes the operation recursive and thus unsafe to optimize.

feedback of scalar value from one loop pass to another

```

for ( i=0; i < 100; i++ )
{
    a[i] = a[i] + s;
    s = a[i]/c[j];
}

```

The variable `s` is used in the first line of the loop to set `a[i]`, and then is set to a function of `a[i]` in the last line. This creates feedback of elements of `a` from one loop pass to the next, which prevents optimization. `s` is called a carry-around scalar, as it carries a value around to the next pass of the loop.

potential feedback of array elements -- use pragma if ok

```

void test(a,b,c)
float *a,*b,*c;
...
for (i=0; i<n; i++)
    *a++ = *b++ + *c++ ;

```

The use of passed-in pointers makes it unclear whether the ranges of locations taken on by `b` and `c` overlap with that of `a`. If you know that they don't overlap use, for example, the assertion level option on invocation (at least `-L1` in this case) to advise VAST-C.

```

for ( i =0; i < 100; i++ )
    a[i+j] = a[i] + b[i];

```

It is not clear whether there is feedback between the two uses of `a` in this loop or not (it depends on the value of `j`). Loops of this kind that you are sure are safe can be optimized by inserting the `nodepchk` pragma in front of the loop. Here is another case where this message would result:

```

for ( i=0; i < 100; i++ )
    b[i] = b[ib[i]] + a[i];

```

`b[ib[i]]` is a gathered array, and as the values in `ib[i]` are unknown, it may conflict with the assignment of elements of `b` on the left side of the equal sign. If the pattern of `b[ib[i]]` is known not to overlap `b[i]`, then the `nodepchk` pragma should be used.

As a convenience, the `-d d` option switch or `nodepchk` pragma with routine or global scope may be used instead of loop-by-loop pragmas.

multiple store conflict

```

for ( i=1; i <= 100; i++ )
{
    if ( c[i] > 0 ) a[i-1] = c[i];
    a[i] = b[i];
}

```

Stores into overlapping sections of the same array must be done in the same order as in the scalar loop.

potential multiple store conflict -- use pragma if ok

```
for ( i =0; i < 100; i++ )  
{  
    a[i+j] = b[i];  
    a[i+k] = c[i];  
}
```

The loop above has a potential overlap between the two stores into a; usually in these situations there is no real overlap between the two sections of a and the `nodepchk` pragma should be used to allow the loop to optimize.

too many data dependency problems

The loop has exceeded the maximum number of data dependency conflicts allowed (10). Optimization of the loop is abandoned at this point to avoid printing out further messages.

8. Decision processes

Allowable conditional constructs

This section describes the varieties of conditional statements in loops and how VAST-C deals with them.

VAST-C can analyze combinations of conditional assignments, conditional and unconditional forward branching and block `ifs`.

Two types of conditions

There are two basic types of conditions, as illustrated in the loops below. The first loop shows a loop independent conditional assignment:

```
for (i=0; i < 100; i++ )
{
    b[i] += 1.0;
    if ( n ) a[i] = b[i] + 2.0;
}
```

The `if` clause (`n`) does not depend on the loop index at all; it remains the same for all iterations of the loop. Thus, we know whether or not `n` is equal to 0 before the loop is executed.

The other kind of condition is shown in this loop:

```
for ( i=0; i < 100; i++ )          (Vectorized)
    if ( c[i] < 0 ) d[i] = e[i] + f[i];
```

It is called loop dependent because the `if` clause (`c[i] < 0`) might change with each loop pass. Sometimes we want to do the calculation (add `e[i]` to `f[i]`) and sometimes we do not.

The %CD field in the loop summary of the VAST-C listing summarizes what percentage of the operations in the translated loop are loop-dependent conditional. A value close to 100 in this field means that the loop is almost completely conditional, and may run slower than the scalar original if the condition is sparse.

For vector loops, you should examine loops with large values in the %CD field to make sure that the calculation in those loops is usually executed. If the calculation is skipped over by conditional statements most of the time, the loop will probably run slower when vectorized, and a novector pragma should be used to turn off vectorization.

Removal of invariant IFs

Invariant ifs are removed from loop nests completely. Code is replicated so that the test is made once outside of the loop and then alternate versions of the loop are executed based on the result of the test.

In removing invariant ifs, VAST-C limits the amount of additional code generated: no more than three additional versions and/or 100 additional lines of code are added.

```
for (i=0; i < 100; i++ )
{
    b[i] = 1.0;
    if ( n ) a[i] = b[i] + 2.0;
    c[i]= a[i] * .5;
}
```

Translation:

```
if ( n )
{
    for (i=0; i < 100; i++ )
    {
        b[i] = 1.0;
        a[i] = b[i] + 2.0;
        c[i]= a[i] * .5;
    }
}
else
{
    for (i=0; i < 100; i++ )
    {
        b[i] = 1.0;
```

```
        c[i]= a[i] * .5;
    }
}
```

Messages about conditionals

The messages below relate to handling of conditional operations.

backward transfers are not optimizable

```
    j = 1;
    for ( i=0; i < 100; i++ )
    {
top:
        a[j] = b[j] + d[i];
        j++;
        if ( b[j] > limit ) goto top;
        c[i] = a[j];
    }
```

The branch to label top is backward, not forward, and prevents optimization. This message appears only if VAST-C was not able to convert the backward branch into a for loop.

branches out of the loop prevent optimization

```
    for ( i=0; i < 100; i++ )
    {
        a[i] = b[i] + c[i];
        if ( a[i] < 0 ) break;
        b[i] = 6.0;
    }
```

branches are too complex to optimize

Because of limits on time and memory, conditional constructs with more than six simultaneously active conditions (i.e. ifs, cases, etc.) are not analyzed.

9. Inline expansion

Introduction

This section describes the inline function expansion feature of VAST-C.

Programs can often receive a performance benefit from the expansion of the bodies of certain functions into the loops which call them. This allows the calling loop as well as the body of the called routine to be optimized. Application codes sometimes have small functions that are called repeatedly; these functions are good candidates for inline expansion. Here is a small example of inline expansion:

Original:

```
{
    ...
    for ( i=0; i < n; i++ )
        a[i] = calc( a[i], x+b[i], 2.0 );
    ...
}
float calc( a,b,c )
float a, b, c;
{
    float t;
    t = a + sqrt( b*b + c*c );
    if ( t < 0 ) t = -( b + c );
    return t;
}
```

Expanding function calc inline:

```

for ( i=0; i < n; i++ )
{
    float calclx,t,blx;
    blx = x + b[i];
    t = a[i] + sqrt( blx*blx + 2.0*2.0 );
    if ( t < 0 ) t = -( blx + 2.0 );
    calclx = t;
    a[i] = calclx;
}

```

Inline expansion reduces function calling overhead. It also allows scalar optimizations such as invariant code relocation and common subexpression analysis to extend across the body of the function as well as the body of the calling loop.

Inline expansion pragmas

autoexpand pragma

The autoexpand pragma is used to invoke automatic routine expansion.

Format:

```
#pragma vd_ autoexpand
```

The *f*, *r*, or *l* scopes can be used. `noautoexpand` cancels the action. The autoexpand pragma with global scope is equivalent to the `-vinline` switch. If you want several levels of expansion from the bottom of the calling tree, use the `-J` option to indicate how many levels of nested inlining you request.

`-vinline` requests only functions at the lowest level be inlined, and is thus similar to `-J1`.

If you want to exempt some functions from autoexpansion, you can use the `-N` invocation parameter. The functions listed after `-N` will not be expanded.

Explicit mode

In explicit mode, routines listed on a pragma are expanded without regard for the automatic mode criteria. If no list is given on the pragma, it directs expansion of all function references in the immediately following statement. This reference need not be inside of a loop.

expand pragma

The expand pragma is supplied for explicit routine expansion.

Format:

```
#pragma vd_ expand [list]
```

Where list is a list of functions to expand in this routine. If (list) is not supplied, expand the next statement.

Scope is ignored on the expand pragma.

Example of pragma use:

```
#pragma vd_ expand (calc)
...
for ( i=0; i < n; i++ )
    a[i] = calc( a[i], b[i]+1, n );
...
```

The -I list switch on the invocation line performs the same function as this pragma, but applies to the entire file. The list is the same as above, but must not be null.

nexpand pragma

Expand the indicated function and all functions it calls. The -Y invocation parameter allows you to request nested expansion from the command line.

search pragma

You can tell VAST-C where to look for the routines to expand with the search pragma.

Format:

```
#pragma vd_ search (filenames)
```

Where filenames is a list of files (separated by commas) in which to search for the routines to be expanded.

If filenames is the special entry `*.c`, then the routine `xyz` will be looked for in file `xyz.c`. (This is the default search method.)

The -S filenames switch on the invocation line performs the same function as this pragma, but applies to the entire file. The filenames are the same as above, and may also be a directory name.

Where to get the code

In order to expand a function, VAST-C needs to know where to find its source. The source location depends on programming style and on the operating system user interface.

Same file

Referenced functions can be searched for in the same file as the calling function ("stacked input"). This necessitates an initial pass by VAST-C through the entire input file (and files included therein) to build a directory of the program units in

the input file. The switch `-e 8` enables this initial pass, which by default is not done.

Explicitly named file

You can supply (via the search pragma) the name of a file in which to search for a particular called routine.

Implicitly named file

In Unix-based systems, C programs are frequently stored such that each routine of the program resides in a separate file with a canonical name (e.g. the name of the function followed by `.c`). This is the default search method.

Possible problems

Separate compilation

C allows program units to be compiled separately and linked together. Because different program units may be compiled at different times it is not possible to completely "fool proof" inline expansion. Even if routines are initially compiled from the same input file, an object of a modified version of a routine could be supplied at a later link.

For example, suppose program A has function B which references function C. Let's say function C is expanded into function B. Later, we decide to change the calculation in function C and so we edit it; rather than recompiling the entire program we just recompile C and link it with the previously compiled routines. Our changes to C are not present in the actual code executed because C is no longer called from B (it was expanded.)

Thus, you must be involved in the routine expansion process at least to the point of knowing which functions must be recompiled when a change is made. For this reason, VAST-C generates both a warning message for every expansion and an expansion event table so that it is clear what has been expanded.

Code size

A problem that may result from inline expansion is code mushrooming; if too many routines are expanded, or the expanded routines are too large, the size of the compiled code may reach unacceptable proportions. This potential problem can be controlled through judicious application of this transformation.

Debugging

Inline code expansion can complicate user run-time debugging; if the program bombs in an expanded section of code, the error is reported in a different routine than the one it originally appeared in.

VAST-C records the original line number of the routine invocation on all the expanded lines.

Compilation rate

Any scheme to analyze more than one program unit at a time will lead to significantly slower compilation rates, and inline expansion is no exception. It may result in two passes over the entire program, and potentially much longer compile times, depending on how much code is brought in line.

Analysis inhibitors

There are several ways in which analysis may be inhibited, thereby prohibiting expansion. An appropriate message informs you of a failed expansion. A full list of messages appears at the end of this section.

Expansion inhibitors

- The function to be expanded can not be located.
- Syntax errors are found in the expansion function.

Inhibitors specific to automatic expansion mode

In automatic mode, all references to functions that meet the following criteria are expanded.

- The function to be expanded has less than 50 lines. This threshold is controllable using the `-Mnnn` switch where `nnn` is the number of lines.
- The routine to be expanded does not call any other external functions that will not also be expanded.
- There are no inhibitors to the expansion.

If these parameters are unsatisfactory, you can always resort to explicit mode. The objective of automatic mode is to catch small, simple external functions. More demanding cases must be explicitly requested.

Note that although called automatic, this mode still requires you to insert a pragma (`autoexpand`) or use a switch (`-Vinline`) when invoking VAST-C. An informational message is issued for each expansion action. This is to remind you that functions that have been expanded into need to be recompiled each time the expanded function is changed.

Inline expansion user messages

All inline expansion messages appear as warnings. VAST-C does not expand any function that has caused the generation of one of the following messages, except for the function expanded message.

Inline expansion summary

VAST-C notifies you of any functions that have been expanded and also informs you as to why a particular function was not expanded. A summary of the routines and all the locations a function was or was not expanded is displayed.

User messages

expansion function not found

The expansion function can not be located, as specified by the search pragma or by the default location.

function not found in input file

The input file has been defined as the location for expansion functions, either by default or via switch. The function is not found in the input file.

expansion function is too big for automatic expansion

The routine has more than the maximum number of lines (default is 50); increase maximum line default or use explicit expansion mode to expand.

external functions found while expanding

The expansion routine references another function beyond the limit for nested expansion; use explicit expansion mode to expand, or increase the -J parameter.

syntax error encountered in expansion routine

The expansion routine has a syntax error and therefore will not be expanded.

exceeded maximum number of expanded functions

The maximum number of functions VAST-C will expand has been exceeded. Currently this number is 600.

function expanded

This warning message will be issued whenever a function has been expanded.

Index

%

%CD field, 44

A

ALIGNED pragma, 18

aligned switch, -Valigned, 7

alignment, 18

ambiguous subscripting, 38

assertion level, 36

assertion levels, 6

assignment operators, 25

associative transformations, 12

autoexpand directive, 47

B

branches out of the loop, 45

C

codespace switch, -Vcodespace, 7

collapse loops, 32

common expressions, 30

complex vectors, 15

conditional statements, 43

D

data dependencies, 34
Data Dependency Conflict, 14
data dependency directives, 38
data dependency messages, 40
disjoint directive, 36, 39
Disjoint pragma, 13
driver, 4

E

expand directive, 47
expansion of functions, 46

F

feedback of array elements, 40
feedback of scalar value, 40
for loops, 22
function arguments, 35
function inlining, 46

G

gather, 19

I

index, 22
indirect addressing, 19, 40
inline expansion, 46
inlining switch, -Vinline, 7
Internal Error, 14
invariant conditionals, 44
iteration count, 19, 24

L

-L switch, 6
least recent load/store switch, -Vleastrecent, 8
loop collapse, 32
loop fusion, 27
loop independent conditional, 43

loop re-rolling, 28

M

messages about conditionals, 45

messages switch, -Vmessages, 8

messages, dependency, 40

multiple store conflict, 41

N

nested expansion, 48

noargumentoverlap switch, -Vnoargumentoverlap, 8

noassoc switch, -Vnoassoc, 8

nodepchk switch, -Vnodepchk, 8

non-contiguous vectors, 19

nopointeroverlap switch, -Vnopointeroverlap, 8

Note Message, 14

NOVECTOR, 12

novector switch, -Vnovector, 8

O

optimizations, 3

options, 8

outer loop unrolling, 32

P

permutation directive, 40

pointers, 21, 39

pointers, and data dependence, 36

potential dependency, 38

potential feedback, 41

R

recurrences, 34

reductions, 19

relation directive, 39

run-time test, 37

S

scalar, 22

scalar division, 31
scatter, 19
search directive, 48
skip switch, -Vskip, 8
statements, 23
strides, 19
structures, 25
switches, 7
Syntax Error, 14

T

Translation Diagnostic, 14
translation diagnostics, 26

U

unrolling, 28
unrolling, outer loops, 32
unrolling, vector, 19

V

vcc driver, 4
vector, 22
vector unrolling, 19
vectorization messages, 8
-Vswitches, 7

W

Warning Message, 14