



Xtensa C and C++ Application Programmer's Guide

For Xtensa® Tools Version 14.08

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Copyright © 2020, 2021 Cadence Design Systems, Inc.. All Rights Reserved
Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522. All other trademarks are the property of their respective holders.

Patents: Licensed under U.S. Patent Nos. 7,526,739; 8,032,857; 8,209,649; 8,266,560; 8,650,516

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

- * The publication may be used solely for personal, informational, and noncommercial purposes;
- * The publication may not be modified in any way;
- * Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement,
- * The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration; and
- * Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

For further assistance, contact Cadence Online Support at <https://support.cadence.com/>.

Product Release:RI-2021.8

Last Updated:12/2021

Modification: 715388

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Tables.....	v
List of Figures.....	vii
Frontmatter.....	ix
1 Introduction to C/C++ Programming on Xtensa Processors.....	11
1.1 Introduction to Xtensa C and C++	13
1.2 Xtensa Compiler Variants	13
1.2.1 Differences between XCC and GCC.....	14
1.2.2 Differences in XT-CLANG.....	15
1.2.3 Common Compiler Options.....	15
1.2.4 Compiler Runtime Libraries.....	18
1.2.5 Changed, Removed or Added XCC and XT-CLANG Features.....	19
1.3 Software Configuration Options.....	20
1.3.1 C and Math Libraries.....	21
1.3.2 Application Binary Interfaces.....	21
1.3.3 Hardware Floating Point ABI.....	22
1.3.4 Build with Reset Handler at Alternate Reset Base.....	22
1.3.5 Use External Reset Base.....	22
1.3.6 RTOS Compatibility Option (Xtensa LX Only).....	23
1.3.7 Target Linux Compatibility Option (Xtensa LX Only).....	23
2 Performance Tuning Problems.....	25
2.1 Diagnosing Performance Problems.....	26
2.2 Types of Performance Problems.....	26
2.2.1 Choosing Algorithms.....	26
2.2.2 Configuration.....	27
2.2.3 Memory System.....	28
2.2.4 Xtensa LX and Xtensa NX processor Simulations and Profiles.....	32
2.2.5 Compiled Code Quality.....	36
3 Performance.....	43
3.1 Controlling Compiler Performance.....	44
3.1.1 Optimizing Functions Individually.....	44
3.1.2 Attributes.....	46
3.1.3 Interprocedural Analysis.....	46
3.1.4 Inlining Functions.....	49
3.1.5 Using Profiling Feedback.....	50
3.1.6 Aliasing.....	55
3.1.7 Loop Pragmas.....	57
3.1.8 SIMD Vectorization.....	59
3.1.8.1 Viewing the Results of Vectorizing Transformations.....	60

3.1.8.2	Aligning Data for Vectorization.....	62
3.1.8.3	Controlling Optimization Through Pragmas.....	65
3.1.8.4	Speculation (XCC Compiler only).....	67
3.1.8.5	Features and Limitations of the Vectorizer.....	68
3.1.8.6	Vectorization Analysis Report.....	69
3.1.8.7	Vectorization Messages.....	72
3.1.9	Software Pipelining.....	72
3.2	General Coding Guidelines.....	74
3.2.1	Avoid Short Scalar Datatypes.....	74
3.2.2	Use Locals Instead of Globals.....	75
3.2.3	Use Arrays Instead of Pointers.....	76
3.2.4	Minimizing Conditionals.....	76
3.2.5	Passing Function Parameters.....	78
3.3	Floating Point.....	79
3.4	C++ Language.....	80
3.4.1	Streams.....	80
3.4.2	Exception Handling.....	81
3.4.3	Run-Time Type Identification (RTTI).....	82
3.4.4	Templates.....	82
3.4.5	Virtual Functions.....	83
3.5	Benchmark Example.....	83
4	Processor Configuration Options.....	87
5	Tensilica Instruction Extension (TIE) Language Programming.....	89
6	Ensuring Software Quality.....	91
6.1	Using C library console and file I/O support for testing.....	92
6.2	Code Coverage.....	94
7	Memory Layout.....	95
7.1	Using Local Memories Only.....	96
7.2	Using Local Memories for Some Code and Data.....	96
8	Devices and Synchronization.....	99
8.1	Memory Ordering.....	100
8.1.1	TIE Ports.....	102
8.2	Cache Coherence.....	104
8.3	Volatile Devices or Memory.....	105
9	Handling interrupts.....	107
9.1	Interrupt and Exception environment.....	108
9.2	Handling exceptions.....	108
9.3	Handling Interrupts.....	110
9.4	Communicating with interrupt handlers.....	110

List of Tables

Table 1: Changed, Removed or Added XCC and XT-CLANG Features.....	19
Table 2: Common Attributes.....	46
Table 3: Inline Option Comparison.....	50
Table 4: Additional XT-CLANG Pragmas.....	58
Table 5: Vectorization Options.....	59
Table 6: Pragmas for Better Optimization.....	67

List of Figures

Figure 1: Setting Memory Latencies in Xplorer.....	29
Figure 2: Cache Explorer Cycles.....	30
Figure 3: Cache Explorer Cycles Using the Link Order Tool.....	31
Figure 4: Data Cache Profile.....	32
Figure 5: Dynamic ISA Profile.....	34
Figure 6: Static ISA Analysis.....	35
Figure 7: Vectorization Assistant.....	40
Figure 8: Auto Vectorization Message.....	42
Figure 9: Replacing Locals With Globals.....	75
Figure 10: Pure Attribute.....	75
Figure 11: Conditional Multiplication.....	77
Figure 12: Unconditional Multiplication.....	77
Figure 13: Conditionals that Can Be Replaced by Lookups.....	77
Figure 14: Lookups Instead of Conditionals.....	77
Figure 15: Base CoreMark Profile.....	84
Figure 16: Adding a Multiplier.....	84
Figure 17: Compiling Using FLIX Configuration.....	85
Figure 18: Optimizing with -O3 Level.....	85
Figure 19: Optimizing with XT-CLANG Compiler.....	86
Figure 20: Communicating with Flags.....	101
Figure 21: S32RI and L32AI.....	101
Figure 22: TIE Ports and Deadlock.....	103
Figure 23: Coherent Use of Flags.....	105

Frontmatter

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- \$ represents your shell prompt, in user-session examples.
- ***literal_input*** indicates literal command-line input.
- *variable* indicates a user parameter.
- *literal_keyword* (in text paragraphs) indicates a literal command keyword.
- *literal_output* indicates literal program output.
- ... *output* ... indicates unspecified program output.
- [*optional- variable*] indicates an optional parameter.
- [*variable*] indicates a parameter within literal square-braces.
- { *variable* } indicates a parameter within literal curly-braces.
- (*variable*) indicates a parameter within literal parentheses.
- | means OR .
- (*var1* | *var2*) indicates a required choice between one of multiple parameters.
- [*var1* | *var2*] indicates an optional choice between one of multiple parameters.
- *var1* [, *varn*]* indicates a list of 1 or more parameters (0 or more repetitions).
- 4 'b 0010 is a 4-bit value specified in binary.
- 12 ' o7016 is a 12-bit value specified in octal.
- 10 ' d4839 is a 10-bit value specified in decimal.
- 32 ' hff2a or 32 ' HFF2A is a 32-bit value specified in hexadecimal

Terms

- 0x at the beginning of a value indicates a hexadecimal value.
- b means bit.
- B means byte.
- flush is deprecated due to potential ambiguity (it may mean write-back or discard).
- Mb means megabit.
- MB means megabyte.
- PC means program counter.
- word means 4 bytes.

1. Introduction to C/C++ Programming on Xtensa Processors

Topics:

- [Introduction to Xtensa C and C++](#)
- [Xtensa Compiler Variants](#)
- [Software Configuration Options](#)

This guide describes the approach to program applications on the Xtensa processors at C and C++ level in the following areas:

- The basics of the tool chain and choice of software configuration options
- Selecting and configuring the processor runtime
- Performance tuning methodologies
- Improving code quality
- How to effectively tune code for high performance and smaller code size
- Adapting applications to different runtime environments
- Moving applications from XCC to XT-CLANG
- How to control memory layout from the C level
- Restricting and controlling memory access
- Developing interrupt and exception handlers in C
- Coordinating and Communicating between processors
- Devices and synchronization

Scope

This guide does not cover system level programming or assembly level programming typically used in system level code. See the *Xtensa System Software Reference Manual* and the *Xtensa Microprocessor Programmer's Guide* for more information on those topics.

Cadence provides and supports both XCC (Xtensa C Compiler), and XT-CLANG compilers for developing C and C++ applications on Cadence processors. More detailed descriptions of these compilers are provided in their respective user's guides: *Xtensa C and C++ Compiler User's Guide* and *Xtensa XT-CLANG Compiler User's Guide*.

This guide does not describe how to compile and run Xtensa target code using cstub files on an x86 platform. For more information, refer to the *Tensilica Instruction*

| *Extension (TIE) Language User's Guide*, chapter
"Simulating TIE Instructions in a Native Environment".

1.1 Introduction to Xtensa C and C++

This is a guide to developing applications that run on Xtensa processors.

This guide describes how to effectively program applications on Xtensa processors at the C and C++ level. Some of the topics covered:

- Selecting compiler toolchain and software configuration options
- Working with the programming environment
- Configuring the processor to suit applications
- Tuning methodologies
- Controlling code and data placement
- Working with devices and other processors
- Handling exceptions and interrupts

1.2 Xtensa Compiler Variants

For historical reasons, Cadence Tensilica IP supports two compiler variants for developing C and C++ applications for Xtensa® processors.

For the latest information on Cadence Tensilica compilers, refer an article on the Cadence Online Support site that addresses this topic: *C/C++ Compiler Variants Supported by the Xtensa® processor*

XT-CLANG is the Xtensa C and C++ compiler available in current releases of Xtensa Software Tools. XT-CLANG is a state-of-the-art C/C++ compiler developed by Cadence, derived from the open source LLVM C/C++ compiler project. (<https://llvm.org/>) XT-CLANG is built on Clang and LLVM, with advanced optimization and code generation, and supports the latest C and C++ standards. The open source LLVM compiler project is under active development, and is continually being extended, enhanced and improved. The LLVM project benefits from a very large developer base, contributing enhancements, optimizations and bug fixes.

Refer to the *Xtensa XT-CLANG Compiler User's Guide* for more details.

The Xtensa C and C++ Compiler (XCC) is the legacy compiler, which is available in older releases of Xtensa Software Tools. This variant is mature and stable but limited in its ability to support future features and modern language standards. XCC represents both the C compiler, called `xt-xcc`, and the C++ compiler, called `xt-xc++`. XCC uses the GCC compiler as a front end. XCC and GCC share the same preprocessor, assembler, and linker. Therefore, usage of XCC is very similar to GCC. They share many of the same options and XCC supports most, but not all GCC extensions. The XCC back end, the phase that performs extensive optimizations and produces assembly code is completely different from the GCC back end.

In releases where XCC is available, refer to the *Xtensa C and C++ Compiler User's Guide* for more details.

This user guide contains information on both XCC and XT-CLANG. XCC or XT-CLANG availability depends on Xtensa tools version. Please check availability in your installed tools location.

Note: XCC compiler is not supported in Xtensa Tools 14.07 and later.

These Xtensa compilers that are used to compile source code generate assembly code and object modules that are generally compatible with each other. They share use of all the various GNU-based command-line tools provided with the Xtensa Toolkit.

Certain features are specific to either XCC or XT-CLANG. Refer to the respective user's guides for information on how these operate for your chosen compiler variant.

For superior modern C++ support, XT-CLANG is recommended. By default, the GNU-based libstdc++11 runtime libraries is used. For the best support of C++11 STL, choose the libc++ libraries by specifying `-stdlib=libc++`. Later specifications like C++14 and C++17 require the `-stdlib=libc++` option.

1.2.1 Differences between XCC and GCC

C++ Exception Handling

One noticeable difference between XCC and GCC is in the use of C++ exception handling. Exception handling adds substantially to code size, whether or not the exceptions are ever used. Therefore, `xt-xc++` has exception handling turned off by default to avoid penalizing programs that do not use exceptions. This is the opposite of standard GCC, which has exception handling enabled. If your program uses exceptions, you can enable exception handling in `xt-xc++` by supplying the `-fexceptions` option on the command line.

Common Block

Another difference between XCC and GCC is the default use of the common block. XCC by default allocates uninitialized, non static, global variables in the `.bss` section. This provides slightly better performance, and enforces the ISO C standard requirement that non static global variables be defined (without the `extern` specifier) only once in a program. GCC, on the other hand, by default allocates uninitialized, non static, global variables in the common block. Using the common block allows the linker to merge multiple such variables of the same name (possibly with one that is initialized), and thus allows building code that relies on the pre-ISO/ANSI ability to define the same variable multiple times. Use the standard GNU option `-fcommon` to get the same behavior as GCC.

Unsigned Char

While not a difference between XCC and GCC, the following is sometimes encountered when compiling code that was only tested on a limited set of processor architectures.

The C and C++ language standards do not define whether the data type `char` is a signed or unsigned quantity. Most implementations choose to make it signed. However, the base Cadence architecture does not have instructions to directly load signed, 8-bit quantities. Instead, an unsigned load followed by a sign extension must be used to implement a signed, 8-bit load. To avoid the additional overhead, in the common case, Cadence defines a `char` to be unsigned. Most applications work either way. However, some applications assume that `char` is signed.

1.2.2 Differences in XT-CLANG

Options, attributes and pragmas

The XT-CLANG command line interface is substantially the same as the LLVM compiler. Options which are equivalent to XCC options remain the same as the original LLVM option. In some cases, the equivalent XCC option is added as a synonym. Also, a number of options were added to XT-CLANG in order to support Xtensa processors. For details, please see the (*Xtensa XT-CLANG Compiler User's Guide*).

See [Changed, Removed or Added XCC and XT-CLANG Features](#) on page 19 for details of specific options that will need attention when porting applications between XCC and XT-CLANG.

Code Generation

The LLVM infrastructure is quite different from that of XCC, so expect differences from XCC in generated code. Performance may also be better or worse depending on the source code and features used.

Note that loop rearrangement is not equivalent; nested loops may need restructuring to achieve a good pipelined schedule. XT-CLANG also handles vectorization differently; some adjustment may be necessary. See [Vectorization](#) for details.

Output

The XT-CLANG compiler generates `.o` files which are compatible with the Xtensa `xt-ld` linker. There are some differences in code structure and section allocation. Such differences are handled transparently during linking.

1.2.3 Common Compiler Options

This section discusses select compiler options for XCC and XT-CLANG

Compiling your application

To compile C source files into .o object files, use the selected compiler driver command that invokes the various phases of the compiler:

```
xt-xcc -c source_files.c
xt-clang -c source_files.c
```

Compile your C++ source files this way:

```
xt-xc++ -c source_files.c
xt-clang++ -c source_files.c
```

To link your application into the final executable:

```
xt-xc++ object_files.o -o executable
xt-clang++ object_files.o -o executable
```

Using the linker, `xt-ld`, directly to link your application is not recommended. Instead, invoke the compiler driver for C++, all necessary libraries for C or C++ are available. The proper environment is also passed through to `xt-ld`. Cadence recommends that you use the XCC or XT-CLANG driver to invoke the linker with the appropriate libraries.

Files suffixed with `.c` are assumed to be C language files. Files suffixed with `.C`, `.cc`, `.c++`, `.cpp` or `.cxx` are assumed to be C++ files. You can also use `xt-xc++` to tell the compiler that a file is C++ rather than C. If you use any C++ files, you must link with `xt-xc++`.

By default, XCC performs no optimization and does not generate debug information needed by debuggers to allow you to symbolically debug your application. Therefore by default, your resultant application will be large, slow, and undebuggable.

To symbolically debug your application, you must use the `-g` option. This option is applicable with or without optimization. Note that, for optimized code the debugger will not print the values of local variables, or any other variables except on function entry, and will jump around as you single step because the optimizer will aggressively interleave code from multiple lines or even multiple functions.

Optimization flags are discussed in more detail in Chapter 3. Briefly, compiler optimization is mainly controlled through the use of the `-Ox` flags. Production code should always be compiled using at least `-O`, which is equivalent to `-O2`. Unoptimized code is usually slower and larger than code compiled with `-O`.

The `-O3` option will cause the compiler to optimize more aggressively, particularly for DSP type code. Performance on average will be a little faster. However, a small number of programs will speed up significantly, up to two times, while others will slow down a little. Most programs will be larger.

The XCC compiler can automatically infer the use of coprocessor instructions from standard C/C++ applications. As an example, the standard C multiplication can be emulated efficiently on the ConnX, Fusion, Vision, or HiFi family of DSPs using their coprocessor multipliers. In addition, most vector DSPs support VLIW execution of many standard C operations. However, some of these coprocessor instructions use coprocessor register files, and it may not always be safe to use those register files, particularly inside of interrupt handlers. Coprocessor register files may be lazily saved and restored using a coprocessor exception mechanism, and some operating systems might not support the use of coprocessor exceptions inside of interrupt handlers.

To ensure safety inside handlers, XCC does not by default infer the use of any instruction that uses any coprocessor state. Nonetheless, application code can benefit from this inference, and this inference can be enabled by compiling with the `-mcoproc` option. Note that the compiler option for vectorization, `-LNO:simd`, also implies `-mcoproc` in addition to enabling vectorization.

The `-Os` flag can be used together with either `-O2` or `-O3`. By itself it implies `-O2`. It instructs the compiler to try to optimize for space rather than for speed. In comparison to `-O2`, code will usually be slower but smaller.

There are many more advanced optimization options, in particular `-ipa` for interprocedural analysis and `-fb_opt` for feedback-directed optimization. These are covered in more detail in Chapter 3 as well as in the *Xtensa C and C++ Compiler User's Guide*. Feedback-based optimization is not supported by the XT-CLANG compiler.

XT-CLANG Compile Options

XCC and XT-CLANG share the same assembler and linker. Command-line options that apply to the compilation phases have the same effect with both compilers. The XT-CLANG C compiler is invoked as `xt-clang` and the C++ compiler as `xt-clang++`.

Options applicable to the front end, such as those for controlling the language dialect or requesting and suppressing warnings, work very similarly in XCC and XT-CLANG.

The XCC option groups for controlling various XCC-specific optimizations (`-INLINE`, `-IPA`, `-LNO`, `-OPT`, `-SWP`) are not supported by the XT-CLANG compiler.

Most of the command-line options that start with `-W`, `-f`, or `-m` have two forms (for example, `-foption` and `-fno-option`).

There are many command-line options which are covered in the relevant compiler user's guide:

- Compiler Output Options - control the type of output XT-CLANG generates.
- Preprocessor Options - control the preprocessor
- Language Dialect Options - control the dialects of C or C++ accepted by the compiler

- Warning Control Options - options for controlling compiler warnings (To request warnings about code constructs that are generally considered questionable, use the -Wall (and for even more checks the -W) option.)
- Debugging Options - control information generated by the compiler to aid you in debugging your application
- Optimization Options - control the level and type of optimizations performed by the compiler
- Inlining Options - control the inliner (XT-CLANG includes a phase that performs function inlining and removal of unused functions.)
- Code Generation Options - control the code generation of the compiler
- Assembler Options - affect the assembly phase.
- Linker Options - control the GNU linker
- Xtensa-Specific Options - dealing with configuration and platform management

1.2.4 Compiler Runtime Libraries

This section discusses how runtime libraries for XCC and XT-CLANG are provided

Choosing C/C++ Standard Libraries for XCC

The standard libraries which support C and C++ development are built when you submit your XPG configuration build. This is true for both software and hardware builds. Certain selections which affect your code are made when editing the configuration.

In addition, you can perform a software upgrade of an existing configuration build and then download a new configuration build. Use this type of XPG build to change your standard library choice and other software options. Using software upgrade does not affect the hardware portion of the build.

There are two choices for C standard library selection: XCLIB and NEWLIB. Note that is choice applies to all compiler variants.

C++ runtime selection

By default, base STL (Standard Template Library) and C++ language support for both XCC and XT-CLANG are provided by libstdc++, the GCC language support. If you use XT-CLANG, then you may choose the LLVM project C++ library, libc++. This is done at compile time by using the --stdlib=libc++ option. Since this affects which header files are included, you should choose one or the other for an entire application.

The --stdlib=libc++ option provides the best modern C++ support. It is the recommended choice for C++11/14/17 development. This option is not available for XCC, and it is removed in Xplorer when XCC is invoked. Care must be taken to not mix these two library choices as it may result in incorrect code execution or compile-time errors

XT-CLANG and XCC are compatible with each other at the object code level for C.

While C++ classes and objects are compatible across both compilers when not using `--stdlib=libc++` option, Cadence recommends selecting one or the other to use exclusively for each executable. Certain complex uses may not work properly and might result in runtime or compile-time errors.

1.2.5 Changed, Removed or Added XCC and XT-CLANG Features

Here is a cross-comparison of features of the two compilers that have differences. This serves as a quick reference for converting your projects from XCC to XT-CLANG. Feature common to XCC and XT-CLANG are not listed here.

Table 1: Changed, Removed or Added XCC and XT-CLANG Features

Feature / Description	XCC	XT-CLANG	Comments
Inter-Procedural-Analysis	-IPA, -ipa	-flto	Not cross-compatible
Super software pipelining option group	-SWP		Not available in XT-CLANG
Create .w2c.c file for compiler transformations	-clist		Not available in XT-CLANG
Save intermediate compiler files	-keep -save-temps	-save-temps	Only -save-temps available in XT-CLANG
Optimization group controls	-OPT	-OPT:memmove_count -OPT:memmove	Most xt-xcc options not available
Allow optimizations that may violate ANSI/IEEE arithmetic rules	-ffast-math	-funsafe-math-optimizations	Suggested substitute, see documentation for details
Enable/disable unsafe optimizations not compliant to IEEE	-fno-unsafe-math-optimizations -funsafe-math-optimizations	-fno-unsafe-math-optimizations -funsafe-math-optimizations	
Autovectorization	-LNO:simd -LNO:simd_v	-LNO:simd -LNO:simd_v -fvectorize	Implementation differs
Control vectorization in Inter-Procedural Analysis		-fno-early-lto-vectorize	Refer to <i>Xtensa XT-CLANG User's Guide</i>
Aligning pointers	-LNO:aligned_pointers=on -LNO:aligned_formal_pointers=on		Not available in XT-CLANG
Inline functions that are (explicitly or implicitly) marked inline.	-INLINE:requested_only -INLINE:requested	-finline-hint-functions-only -finline-hint-functions	

Feature / Description	XCC	XT-CLANG	Comments
Inline functions based on heuristics	-INLINE:aggressive=off -INLINE:aggressive=on	-finline-functions	-mllvm -inline-threshold, -inlinehint-threshold, -inline-cold-callsite-threshold, and -inline-inst-number-threshold
Control generation of floating-point multiply/add rather than separate multiply and add operations	-mfused-madd -mno-fused-madd	-mfused-madd -mno-fused-madd -ffp-contract=on -ffp-contract=off -ffp-contract=fast	
Attribute alignment for the function, variable or structure field being declared	__attribute__((aligned (n)))	__attribute__((aligned(n))) __attribute__((align_value(n)))	
Alias avoidance	__restrict__ -fstrict-aliasing -fno-strict-aliasing -OPT:alias=VAL VAL=any,typed,restrict,disjoint	__restrict__ -fstrict-aliasing -fno-strict-aliasing	
Super Software Pipelining	#pragma super_swp ...	#pragma super_swp ... [does nothing]	Parsed but ignored in XT-CLANG
Optmization per function	_attribute__((optimize ("-O0")))	__attribute__((optimize ("-O0")))	-Os command line option in XT-CLANG cannot be overridden
File-based optimization control	-fopt-use -fopt-gen		Not available in XT-CLANG
Compiler Messages			XT-CLANG is generally more strict, more warnings

1.3 Software Configuration Options

Configuration options which only affect the target software

Software configuration options require a new processor configuration, but do not affect the generated hardware. Therefore, options can be selected after a hardware design has been

completed. The initial choices set in the software pane of the processor generator when creating a configuration will be used to generate matching diagnostics with the HW package.

You can easily explore alternatives by building the HW + SW one way, and then building "software upgrades" of the original configuration with different combinations of target software options. "Upgrades" can be variants built with the same XPG release, or they can be built with newer XPG releases.

1.3.1 C and Math Libraries

Cadence offers the choice of two C and math libraries: `newlib` from Red Hat, Inc. and the Xtensa C library (`xclib`).

Use the Software tab of the XPG editor to select a C standard library. This tab is shown when you edit your build configuration through the Xtensa Processor Generator.

- The `xclib` has similar performance to `newlib` and is smaller. It strictly implements the C library as defined by the C standard and hence may not implement all the extensions supported by `newlib`. The philosophy of the library is standards compliance and simplicity. So, for example, the `malloc` routine is simple and hence fast but might cause more memory fragmentation on programs that extensively `malloc` and `free`. The Xtensa C library places no open source restrictions *on the C user (there are minor restrictions for the C++ user)*.
- The `xclib` may have other desired features. For some DSPs, optimized math functions from the standard C library `math.h` are integrated into it. For example, the Fusion, HiFi, and ConnX DSP families have this feature.

Review your contract or the files in the `XtensaTools/misc` directory for details about the various licensing requirements.

1.3.2 Application Binary Interfaces

Cadence offers the choice of two Application Binary Interfaces (ABIs) for Xtensa NX and Xtensa LX processors: the windowed ABI and the CALL0 ABI. With the windowed ABI, each function call (that is, a CALL8 instruction) rotates the Xtensa register windows and thereby immediately gives the called function a set of extra scratch registers. By default, Xtensa tools use the CALL8 instruction if available. Without the windowed ABI, each function call is implemented using a CALL0 instruction and the compiler must typically save and restore to memory scratch variables used by the callee. Application code compiled using CALL0 is typically larger than application code compiled using the windowed ABI. Performance of loop-intensive code is marginally slower with CALL0 while more call intensive code is slower.

Given these characteristics, the windowed ABI is most popular. However, there are also advantages to the CALL0 ABI. The CALL0 ABI enables hardware configurations with only 16 AR registers, thereby allowing significantly smaller hardware configurations. Interrupt and context switching latency is lower with CALL0 than with the windowed ABI. Using the CALL0 ABI, you can manually rotate the register files in a single cycle in special code or interrupt handlers for faster specialized context switching.

An application cannot mix the two ABIs. However, it is possible to use the windowed ABI for an application and CALL0 for certain high priority interrupts. The use of CALL0 in this context enables interrupt handlers to be written in C without the higher overhead of saving and restoring all the AR registers.

Related Links

[*AR Registers Count*](#)

1.3.3 Hardware Floating Point ABI

Cadence offers two familiar scalar floating point options: single or double precision following the IEEE 754 standard; also several coprocessors optionally include scalar or vector floating point hardware capabilities featuring half, single, and double precision. For more details, refer to the specific DSP user's guide for each of the HiFi, Fusion, Vision, ConnX and FPX DSP families.

The coprocessors that include scalar floating point capabilities implicitly use the Floating Point ABI which makes use of floating point registers for function calls.

The familiar single and double precision options use the standard ABI by default. The standard ABI makes use of AR registers for FP value passing. The Hardware Floating Point ABI is chosen if the configuration hardware supports it. This leads to more efficient passing of floating point variables, especially for double precision. Exercise caution when integrating with object code compiled for other configurations; the standard and hardware FP ABI's are not compatible.

1.3.4 Build with Reset Handler at Alternate Reset Base

For processor configurations that support relocatable vectors, at configuration time, a primary and alternate "static vector group base address" can be configured. This address is a base from which the reset vector and memory error vector (if configured) are offset. Which address (primary / alternate) is used at processor reset is controlled by an input pin which can be asserted to select the alternate base. By default, the software configuration build will assume the primary static base is used, and will generate reset code for those addresses. This option chooses whether to build software with the reset code at the alternate address.

1.3.5 Use External Reset Base

Normally, XTOS is built with the reset handler at its primary configured location. Selecting this option causes the reset to be built at the alternate reset address, which must match the external reset address asserted on the extreset pins

For processor configurations that support relocatable vectors, at configuration time, a primary "static vector group base address" is configured by filling in the reset field in the vectors. This address defines reset vector and the memory error vector (if configured) is offset from this address. An Alternate reset address must also be defined. This is the reset address by default when an application is linked with "alternate reset" specified. With external reset, this must also match the address asserted at reset on the extreset pins.

Which address (primary / external) is used at processor reset is controlled by an input pin, alternate reset. When not asserted, the primary reset address is used. When alternate reset is asserted, then the extreset pins specify the reset address.

An XPG build can use either the primary or the alterate reset address, selected when you are uploading. To use any other address the LSP must be modified to place the reset vector at the location asserted on the external reset pins.

1.3.6 RTOS Compatibility Option (Xtensa LX Only)

For Xtensa LX configurations, Generic RTOS Compatibility: ensures selection of a set of features required by many RTOSes

This option does not have any direct effect on processor software or hardware; it is a compatibility checking option to help avoid configuration omissions that might have later impact on what software can run on the processor.



Note: This option is for Xtensa LX only. Contact Cadence if you want to use this option.

1.3.7 Target Linux Compatibility Option (Xtensa LX Only)

For Xtensa LX configurations, this option ensures the minimum requirements for running Linux on Xtensa are met.

This option does not have any direct effect on processor software or hardware; it is a compatibility checking option to help avoid configuration omissions that might have later impact on what software can run on the processor.

Xtensa processors support very varied memory maps with flexible combinations of local and system memories and caches. However, running Linux on an Xtensa processor places several significant restrictions on the memory subsystem (for example, the Full MMU with TLBs and a memory layout that is conducive to an appropriate virtual memory layout).

If you plan to run Linux on an Xtensa processor, it is strongly recommended that you start with one of the available sample processor templates, which sets up an appropriate memory configuration.



Note: This option is for Xtensa LX only. Contact Cadence if you want to use this option.

2. Performance Tuning Problems

Topics:

- *Diagnosing Performance Problems*
- *Types of Performance Problems*

This section details the tools and methodologies for maximizing application performance and minimizing code size on Cadence processors.

Subsections describe the methodologies in more detail.

2.1 Diagnosing Performance Problems

Without tools, it can be difficult to know where your performance problems are or what you can do to solve them. For example, a floating-point multiplication using software emulation might take approximately 100 cycles. If 1% of the operations in your application are floating-point multiplies, the multiplication routine represents significant computational overhead. If 0.01% of the operations are floating-point multiplies, the overhead might be negligible.

There are multiple types of tools that can help you diagnose software-performance problems. Profiling, available in simulation or on real hardware through Xtensa Xplorer, as well as through `xt-gprof`, shows you where your application code is spending the most time, either at the function level or at the source-line or assembly-line level. You should concentrate your efforts on regions that consume a significant portion of the processor's or task's time. In addition, the tools are able to break down the time spent executing an application event-by-event. For example, they can tell you how much time is spent globally servicing cache misses or accessing uncached memory, and they can tell you which regions of code suffer the most from those events. This information helps you know whether to concentrate your efforts on generated code sequences or on your memory subsystem. Similarly, the ISA profiler can tell you how frequently every instruction or operation is executed, allowing you to discover if a selected hardware configuration or TIE instruction is being effectively used.

Software development tools also give static information useful in tuning applications. Measuring code size on a function by function basis is useful when trying to minimize memory usage. Looking at the compiler generated assembly file, the `.s` file, the compiler inserts comments to aid in tuning. When vectorizing to try to take advantage of a ConnX, Vision, Fusion, Floating Point (FPX) or HiFi DSPs, the compiler can also generate analysis messages explaining what issues were encountered in the code. The messages are enabled on the command line with the `-LNO:simd_v` option or in Xplorer's *Vectorization Assistant* tool. The Xplorer also annotates the software pipeline and vectorization reports in the C code perspective.

2.2 Types of Performance Problems

When you know where to start, you can begin to tune your code. There are several aspects to performance including algorithmic, configuration, memory system, microarchitectural, and compiler code quality, as discussed in this section.

2.2.1 Choosing Algorithms

Choice of algorithm has a first-order effect on performance, code size and power. A radix-4 FFT fundamentally has fewer multiplies than a radix-2 FFT. A heap sort fundamentally requires fewer compares than an insertion sort. Algorithm choice is mostly independent of

processor architecture, and there are few tools to help select among alternative algorithms. However, there is some connection between algorithm and processor or system architecture.

For example, a radix-4 FFT requires fewer multiplies than a radix-2, and a radix-8 FFT requires fewer multiplies than either of the other two types. However, as you increase the FFT radix, you increase the operation complexity and the number of registers required. At sufficiently high radix, performance becomes dominated by overhead and data shuffling. A typical Xtensa processor configuration tends to handle radix-4 better than either radix-2 or radix-8. Be mindful of your architecture when choosing algorithms and, for the Xtensa processor, be mindful of your algorithm when tuning the architecture.

2.2.2 Configuration

The choice of processor configuration can greatly affect performance. The use of TIE can increase performance by many factors, but configuration parameters and choice of memory subsystem can also make a substantial difference. Specific details of different configuration options are discussed in the *Xtensa® LX Microprocessor Data Book* *Xtensa® NX Microprocessor Data Book*. Specific details about the TIE language are discussed in *Tensilica® Instruction Extension (TIE) Language User's Guide*.

There are two basic techniques that can be used to improve your choice of core processor configuration.

To use the first technique, profile your application and take a close look at the hot spots. If the profiling shows that much time is spent in an integer multiplication emulation routine, then adding a multiplier to your configuration will probably help.

Using the second technique, build multiple configurations and see how performance of your application varies across configurations. Xtensa Xplorer allows you to manage multiple configurations and graphically compare the performance of your application using different configurations.

For example, the Xtensa Xplorer can be used to compare the integer multiply acceleration using two configurations: one with no hardware multipliers, and one with a MUL16, an option that adds 16-bit multiplication in hardware. The cycle performance is expected to significantly improve when the MUL16 is added.

Similarly, the Xtensa Xplorer can also be used to compare the performance of an application performing floating-point multiplies using two configurations: one without the Xtensa floating-point unit (FPU), and one with the FPU. As with integer multiply acceleration, the hardware acceleration for FPU helps significantly reduce the cycles needed for your application.

The Xplorer has various tools that can be used to profile the application under study using various configurations following the steps shown below:

- Profile each configuration run of interest and provide a different name for each.
- From the Benchmark Perspective, in the Benchmark Results tab, Select Profile Data Sets to select the various profile runs
- Select **Create New Custom Chart** to create a chart to compare the various sets

Several graph display modes are available, and can be selected using the buttons at the upper-right of the comparison window. This allows you to experiment with the various available configuration options and decide if the added hardware provides enough acceleration.

2.2.3 Memory System

In modern embedded systems, the speed with which you can get data into and out of the processor can often be the most critical aspect of the processor's performance. Xtensa processors give you freedom in designing external interfaces. Communication can be through local memory ports, the Xtensa Local Memory Interface (XLMI), the Xtensa Processor Interface (PIF), AHB, AXI, TIE ports, TIE queue interfaces, TIE lookups, or interrupt lines. Each of these interface types can be tailored for the application. You can, for example, choose the width of the PIF, the size of caches (if any), and the number of memory units.

Cadence provides several tools to measure and analyze memory-system performance in simulation. The standalone ISS simulator accurately models caches and the interactions between the memory system and the pipeline for a fixed-latency/fixed-bandwidth memory system. The standalone ISS allows you to independently set read and write latencies as well as block repeat rates using the flags `--read_delay`, `--read_repeat`, `--write_delay` and `--write_repeat`. On a block read transaction, a cache miss for example, the first PIF sized chunk of data is assumed to arrive `--read_delay+1` cycles after the address is presented to the PIF. Each subsequent PIF sized chunk of data arrives `--read_repeat+1` cycles later. These flags are also settable from the Run dialog box in Xplorer as shown in the figure below.

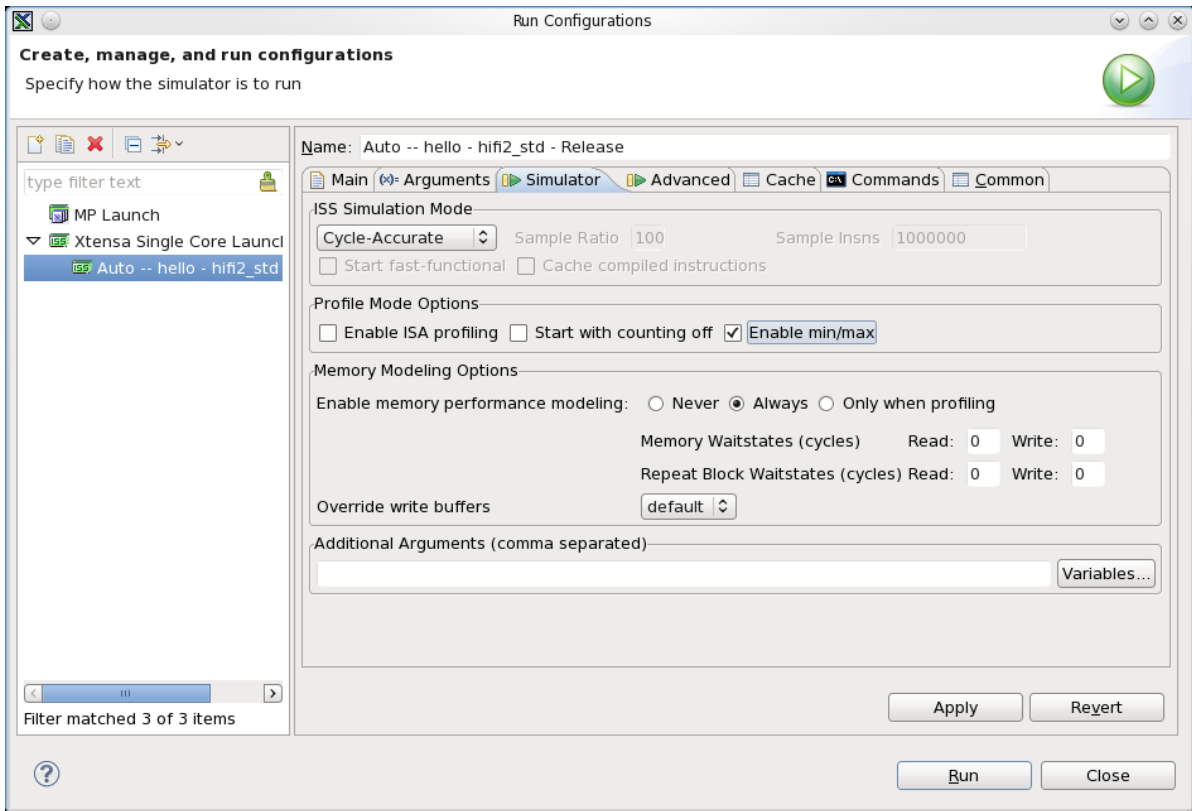


Figure 1: Setting Memory Latencies in Xplorer

When creating shared memory subsystems in Xplorer, the memory delays can also be set when partitioning the subsystem memories.

For more complicated memory modeling, Cadence provides XTSC, and SystemC based simulation libraries, that allow you to build accurate models of more complicated memory systems including DMA, TIE queue interfaces, ports and lookups, and custom devices and interconnects.

By default, the standalone uniprocessor ISS does not simulate the memory subsystem; it assumes zero delay cycles on all loads and stores. This policy allows you to tune the other aspects of performance independently of your memory subsystem. Particularly with caches, small changes that improve microarchitectural performance might randomly degrade memory-system performance as they change data or instruction layout. Nonetheless, all these small changes in aggregate will probably improve performance. Simulations with an ideal memory system allow you to ignore these effects until you are ready to tune your memory system. After architectural tuning of the processor is finished, you can select memory modeling in the stand-alone ISS simulation, or use the XTSC simulation environment to add memory-delay characteristics to the simulation.

As with other aspects of configuration, Xtensa Xplorer allows you to easily compare performance of an application using multiple memory systems. The Cache Explorer feature of Xtensa Xplorer allows you to graphically select a range of cache parameters and have the system automatically simulate and compare performance for all selected configurations. To illustrate the use of this tool, an MP3 decoder running on a series of Cadence HiFi 3 configurations was tested.

The control panel is accessible via the Tools/Cache and Memory Explorer menu in Xplorer. In this example, all combinations of direct mapped and 2-way set associative 4K and 8K Icache sizes together with all combinations of direct mapped and 2-way set associative 8K and 16K Dcache sizes are to be evaluated.

The results are shown in the figure below. The miss penalties are significant, and the data cache misses are larger than the instruction cache misses. For the data cache misses, every fourth bar is significantly larger than the others. This corresponds to an 8K, direct mapped data cache. It is likely necessary for this application to either increase the data cache size to 16K or use a set-associative cache. The first four results correspond to a 4K direct-mapped instruction cache. The instruction cache penalties are significantly higher for these configurations.

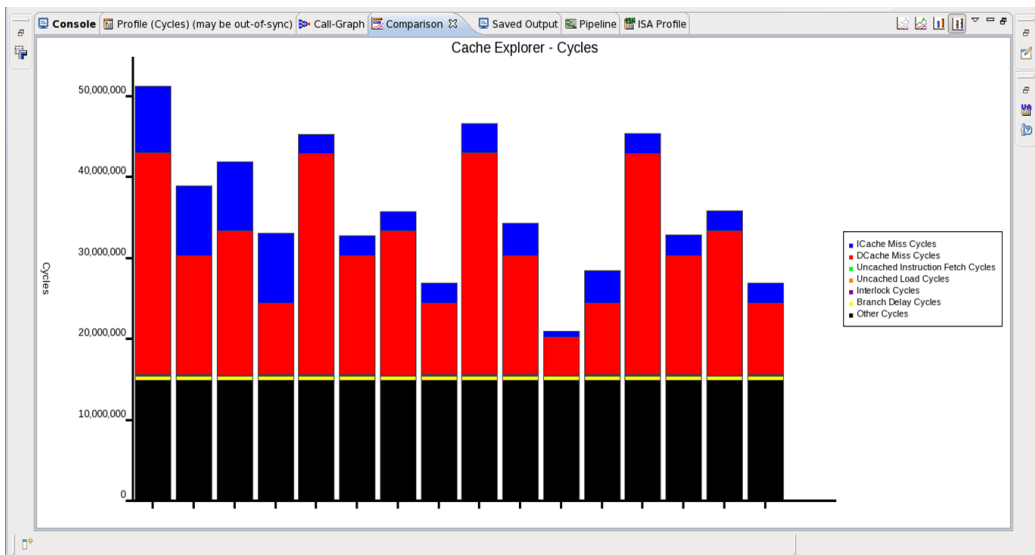


Figure 2: Cache Explorer Cycles

The Cache Explorer is a good tool for helping you decide which cache configurations to select, and it is also useful for software tuning. For example, we see from the tool that I cache misses are a problem by default when the I-cache is direct mapped and only 4 KB. Xplorer provides a software tool to rearrange the order of functions in an executable to minimize I cache misses. To invoke the tool, first profile the application. Then right-click the project and select *Update Link Order*. Choose one or more profile files to guide the reordering. These

profile files should be representative of typical execution runs. Typically one should either use a profile file that measures cycles or one that measures cycles due to instruction cache misses. Once the link order file has been generated, change the project build properties to use the link order file as part of the link step. The results of using the tool on the MP3 application are shown in the figure below. The penalty for instruction cache misses has been significantly reduced, allowing the use of the direct-mapped 4K I cache.

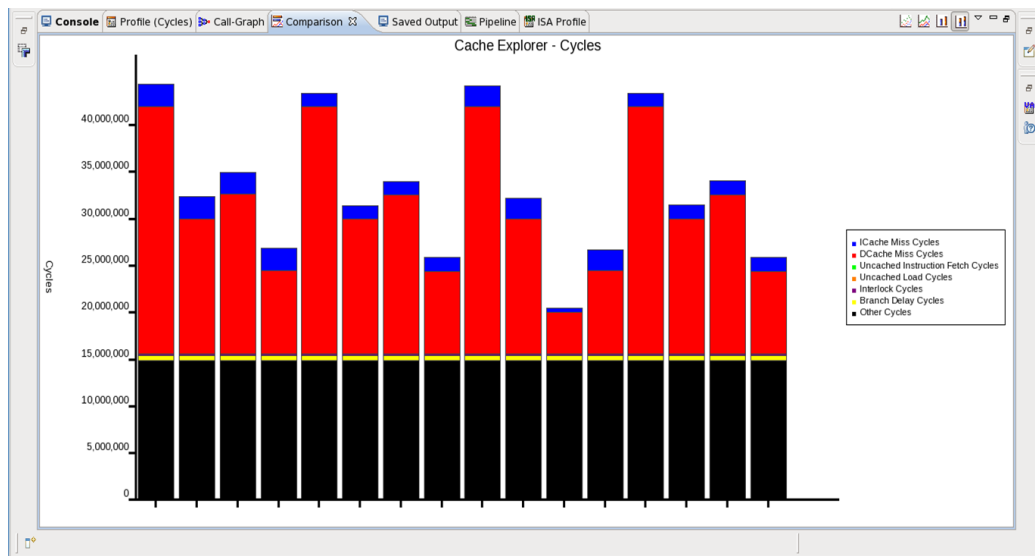


Figure 3: Cache Explorer Cycles Using the Link Order Tool

The Cache Explorer is closely tie together with the profiling tools, it allows you to easily focus in on problematic portions of your application. For each evaluated configuration, you can easily see the corresponding profile of your application. In addition to the traditional time based profiling, you can also profile based on events such as cache misses. Consider again the MP3 example. The performance of the direct-mapped 8K data cache is poor. [Figure 4: Data Cache Profile](#) on page 32 shows a data cache profile of that configuration. From the profile, you can see that over 40% of the data cache misses are in a single C function, and the vast majority of those misses are in one loop. Direct-mapped caches perform poorly relative to set associative caches when multiple memory references conflict with each other. Looking at the single loop, you see that it is streaming through two different arrays. Perhaps those two arrays line up in the same cache lines. As a quick experiment, some padding was randomly added to the data memory. The number of data cache misses in the function was reduced by 45% and overall performance improved by 5%.

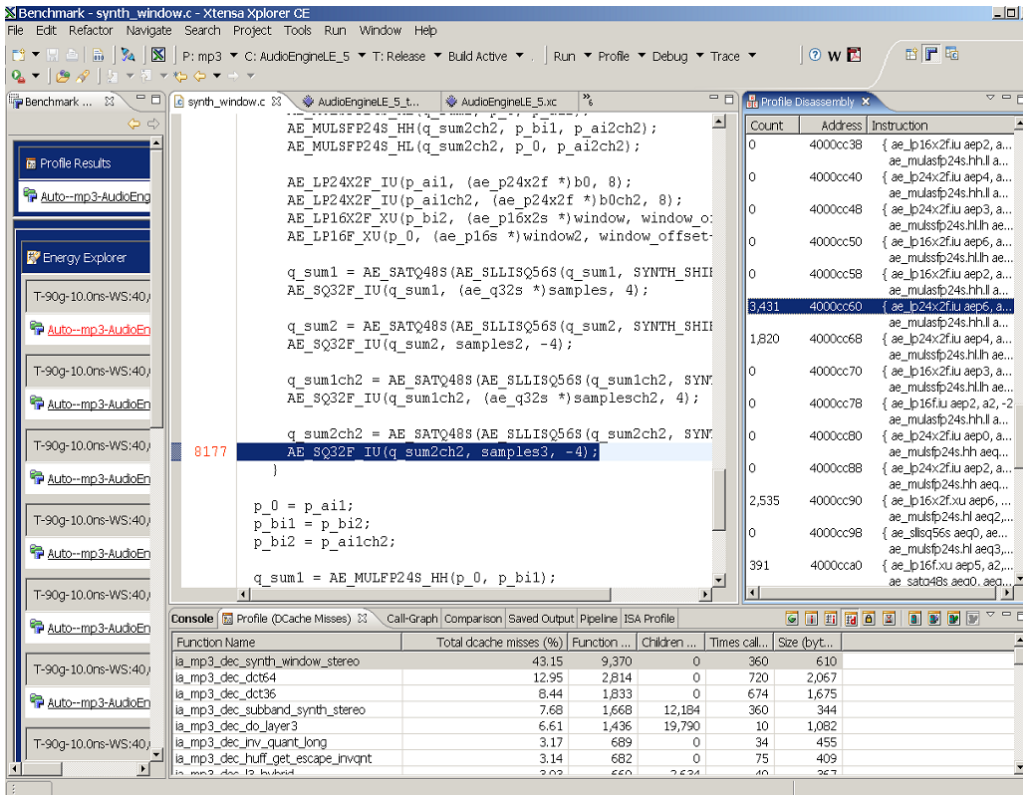


Figure 4: Data Cache Profile

Finally, when tuning your memory subsystem, it is important to model a system and application similar to the one you actually will use. For example, on a system with local memories, DMA might significantly improve performance for some applications. Simulating such applications without modeling DMA might lead one to incorrect conclusions about the required cache parameters.

2.2.4 Xtensa LX and Xtensa NX processor Simulations and Profiles

The XCC and XT-CLANG compilers generate efficient code for the full range of Tensilica processor configurations. Tensilica processors are based on Xtensa LX or Xtensa NX configurations. This section summarizes some differences in the ISS summary and profiles for Xtensa LX and Xtensa NX processors.

Xtensa LX Pipeline

At full speed, the Xtensa processor's pipeline can execute one instruction every cycle. However, various situations can stall the pipeline. The most common reasons for stalling the processor are memory delays, branch delays, and pipeline interlocks. Other reasons are

described more fully in the *Xtensa Instruction Set Simulator (ISS) User's Guide*. The ISS summarizes the number of cycles stalled for different reasons. The following shows sample output from an ISS summary:

```
Cycles: total = 467284
```

		CPI	Summed CPI	percent	Summed Cycle	percent Cycle
Committed instructions	447814	(1.0000	1.0000	95.83	95.83)
Taken branches	3140	(0.0070	1.0070	0.67	96.51)
Pipeline interlocks	47	(0.0001	1.0071	0.01	96.52)
ICache misses	455	(0.0010	1.0081	0.10	96.61)
DCache misses	1999	(0.0045	1.0126	0.43	97.04)
Exceptions	15	(0.0000	1.0126	0.00	97.04)
Uncached ifetches	9252	(0.0207	1.0333	1.98	99.02)
Uncached loads	61	(0.0001	1.0334	0.01	99.04)
Sync replays	1985	(0.0044	1.0379	0.42	99.46)
Special instructions	510	(0.0011	1.0390	0.11	99.57)
Loop overhead	2001	(0.0045	1.0435	0.43	100.00)
Reset	5	(0.0000	1.0435	0.00	100.00)

In addition, the profiler allows you to create profiles based on events, such as branch penalties. This feature allows you, for example, to identify functions, lines, or instructions in your program that suffer the most from branch penalties.

In many situations, branch penalties are a bigger problem than interlocks. Except for zero-overhead loops, every aligned branch taken in the 5-stage version of the processor's pipeline suffers a 2-cycle branch penalty. One additional penalty cycle is required for unaligned branch targets (the instruction at the branch target address is not aligned so that the entire instruction fits in a single, 32-bit-aligned fetch boundary for a 32-bit-wide instruction fetch or a single, 64-/128-bit aligned fetch boundary for a 64-/128-bit instruction fetch). A 7-stage version of the Xtensa pipeline adds one additional branch-penalty cycle.

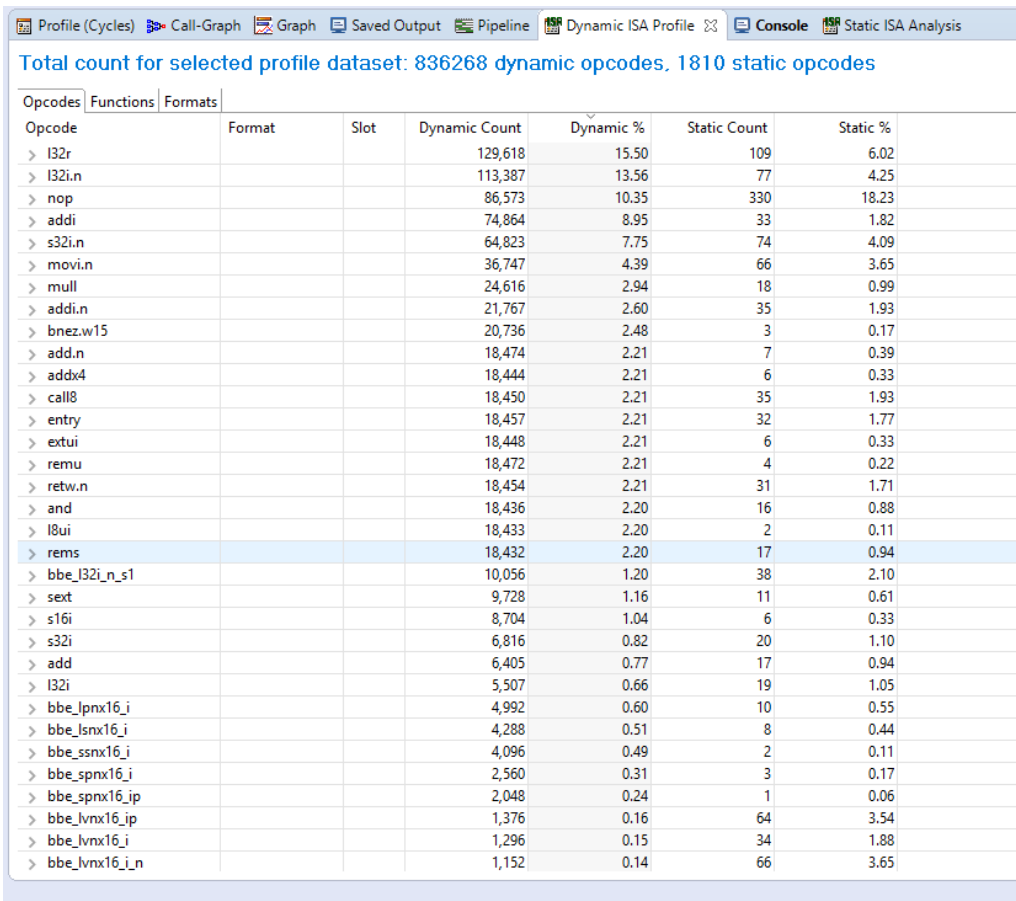
Note: If the branch is not taken, there are no branch-penalty cycles.

Branch penalties can be mitigated or reduced by judicious use of compiler optimizations such as the use of profiling feedback. This is covered in more detail in Section 3.1.4. When compiling for speed rather than space, the tool-chain might replace a 16-bit instruction with an equivalent 24-bit instruction or might add padding to unexecuted regions of code in order to better align branch targets. Penalties can also be mitigated through processor configuration choices. A 5-stage configuration will suffer significantly fewer branch delays than a 7-stage one. A configuration with a 64- or 128-bit fetch will suffer a little less from branch delays. Additionally, the range of branch immediates in the base 24-bit Cadence architecture is limited. Branch delays can be significantly reduced by creating wider branch instructions in TIE that utilize larger immediates. Such branches are standard with many Xtensa DSPs and can be easily added to any FLIX configuration by the Xtensa TIE developer. See the *Tensilica Instruction Extension (TIE) Language Reference Manual* for details.

The output of the ISS also shows you the percentage of cycles spent in source interlocks. The pipeline interlocks whenever a result takes multiple cycles to compute, and the compiler

is not able to schedule meaningful other instructions between the instruction that defines a result and the one that consumes it. Interlocks might be fundamental; some code does not have meaningful parallelism. Some might be a function of the configuration; a 7-stage pipe may have significantly more load-use interlocks. They might also be caused by non-optimal scheduling from the compiler. The latter will be covered in more detail in Chapter 3.

With FLIX (VLIW), the summary figure for interlocks might only show part of the problem. Non-optimal scheduling might not cause the processor to stall, but might prevent the compiler from filling all the VLIW slots. In such scenarios, some slots might be filled by NOP operations. The Dynamic ISA Profile is a good tool for seeing how many NOP operations are dynamically executed by the processor. The Dynamic ISA Profile can be used as a standalone command line tool or as a tab in the Profile tool in Xplorer. The results of the Dynamic ISA Profile for an example Xtensa LX configuration are shown below. Note that dynamic profile can be seen per opcode, function, or FLIX format for complete analysis of the code executed.

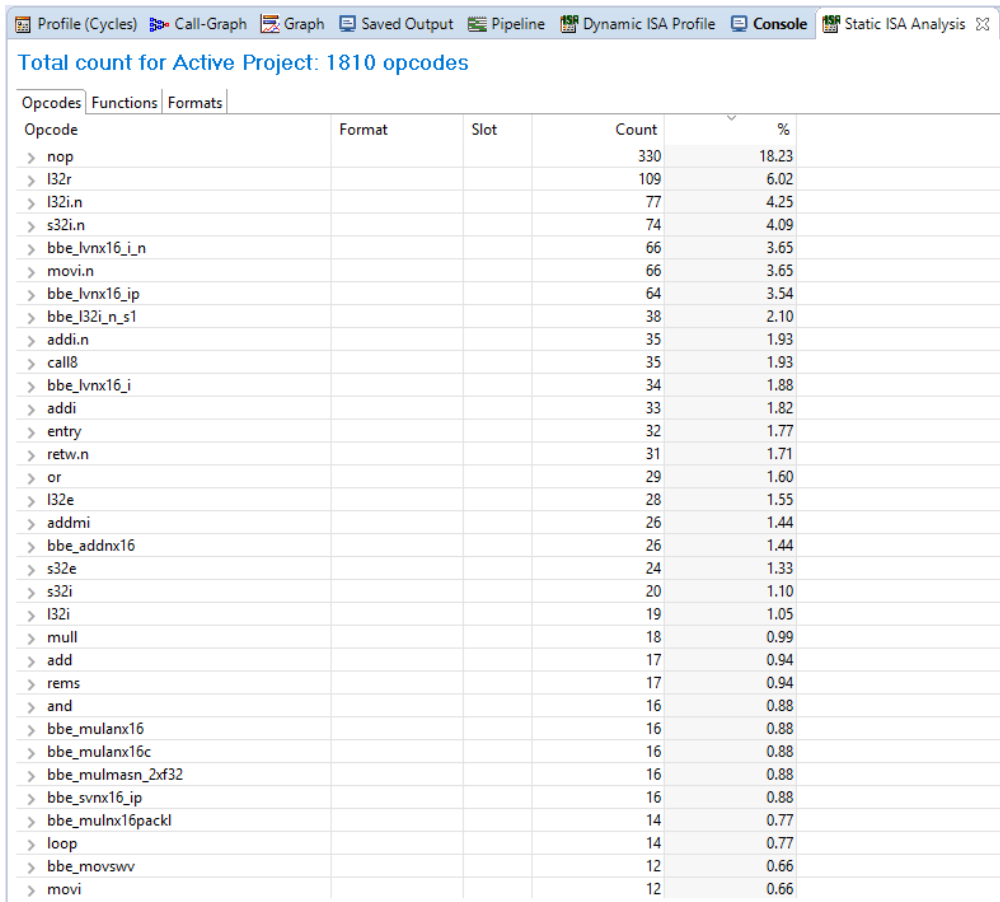


The screenshot shows the 'Dynamic ISA Profile' tab in a software interface. At the top, there are tabs for 'Profile (Cycles)', 'Call-Graph', 'Graph', 'Saved Output', 'Pipeline', 'Dynamic ISA Profile' (selected), 'Console', and 'Static ISA Analysis'. Below the tabs, a summary line reads: 'Total count for selected profile dataset: 836268 dynamic opcodes, 1810 static opcodes'. The main content is a table with columns: 'OpCodes', 'Functions', 'Formats', 'Opcode', 'Format', 'Slot', 'Dynamic Count', 'Dynamic %', 'Static Count', and 'Static %'. The table lists various opcodes such as l32r, l32i.n, nop, addi, s32i.n, movi.n, mull, addi.n, bnez.w15, add.n, addx4, call8, entry, extui, remu, retw.n, and and, along with their dynamic and static counts and percentages. The 'remu' opcode is highlighted in blue.

OpCodes	Functions	Formats	Opcode	Format	Slot	Dynamic Count	Dynamic %	Static Count	Static %
>			l32r			129,618	15.50	109	6.02
>			l32i.n			113,387	13.56	77	4.25
>			nop			86,573	10.35	330	18.23
>			addi			74,864	8.95	33	1.82
>			s32i.n			64,823	7.75	74	4.09
>			movi.n			36,747	4.39	66	3.65
>			mull			24,616	2.94	18	0.99
>			addi.n			21,767	2.60	35	1.93
>			bnez.w15			20,736	2.48	3	0.17
>			add.n			18,474	2.21	7	0.39
>			addx4			18,444	2.21	6	0.33
>			call8			18,450	2.21	35	1.93
>			entry			18,457	2.21	32	1.77
>			extui			18,448	2.21	6	0.33
>			remu			18,472	2.21	4	0.22
>			retw.n			18,454	2.21	31	1.71
>			and			18,436	2.20	16	0.88
>			l8ui			18,433	2.20	2	0.11
>			remu			18,432	2.20	17	0.94
>			bbe_l32i_n_s1			10,056	1.20	38	2.10
>			sxt			9,728	1.16	11	0.61
>			s16i			8,704	1.04	6	0.33
>			s32i			6,816	0.82	20	1.10
>			add			6,405	0.77	17	0.94
>			l32i			5,507	0.66	19	1.05
>			bbe_lpnx16_i			4,992	0.60	10	0.55
>			bbe_lsnx16_i			4,288	0.51	8	0.44
>			bbe_ssnx16_i			4,096	0.49	2	0.11
>			bbe_spnx16_i			2,560	0.31	3	0.17
>			bbe_spnx16_ip			2,048	0.24	1	0.06
>			bbe_lvnx16_ip			1,376	0.16	64	3.54
>			bbe_lvnx16_i			1,296	0.15	34	1.88
>			bbe_lvnx16_i_n			1,152	0.14	66	3.65

Figure 5: Dynamic ISA Profile

An additional useful profile tool is Static ISA Analysis. It provides the breakdown of opcode, function, and FLIX format counts without profiling the code. An example is shown below. It is simply a distribution of your instructions that are part of your program without considering execution flow.



Total count for Active Project: 1810 opcodes

Opcode	Format	Slot	Count	%
> nop			330	18.23
> l32r			109	6.02
> l32i.n			77	4.25
> s32i.n			74	4.09
> bbe_lvn16_i.n			66	3.65
> movi.n			66	3.65
> bbe_lvn16_ip			64	3.54
> bbe_l32i_n_s1			38	2.10
> addi.n			35	1.93
> call8			35	1.93
> bbe_lvn16_i			34	1.88
> addi			33	1.82
> entry			32	1.77
> retw.n			31	1.71
> or			29	1.60
> l32e			28	1.55
> addmi			26	1.44
> bbe_addnx16			26	1.44
> s32e			24	1.33
> s32i			20	1.10
> l32i			19	1.05
> mull			18	0.99
> add			17	0.94
> rems			17	0.94
> and			16	0.88
> bbe_mulanx16			16	0.88
> bbe_mulanx16c			16	0.88
> bbe_mulmasn_2xf32			16	0.88
> bbe_svn16_ip			16	0.88
> bbe_mulnx16packl			14	0.77
> loop			14	0.77
> bbe_movswv			12	0.66
> movi			12	0.66

Figure 6: Static ISA Analysis

Xtensa NX Pipeline

Compared to the Xtensa LX pipeline, the Xtensa NX pipeline is a longer 11-stage pipeline to enable higher clock speeds. The summary the ISS provides after profiling is similar to the LX case but with some different categories to more explicitly categorize each cycle execution. Following is a sample output from an ISS summary for NX configuration profile:

```
Cycles: total = 1424989
Summed |          Summed
```

CPI	CPI	% Cycle		% Cycle			
Committed instructions	808365	(1.0000	1.0000	56.73	56.73)	
Mispredicted branches	302420	(0.3741	1.3741	21.22	77.95)	
Predicted branch stalls	8755	(0.0108	1.3849	0.61	78.56)	
Pipeline interlocks	299878	(0.3710	1.7559	21.04	99.61)	
Memory RAW stalls	51	(0.0001	1.7560	0.00	99.61)	
Special instructions	1860	(0.0023	1.7583	0.13	99.74)	
Exceptions	3540	(0.0044	1.7627	0.25	99.99)	
Sync replays	110	(0.0001	1.7628	0.01	100.00)	
Reset	10	(0.0000	1.7628	0.00	100.00)	

You can still create profiles based on events, such as branch penalties. For example, this feature allows you to identify functions, lines, or instructions in your program that suffer the most from branch penalties. Note that in NX generation branch prediction is available. This typically improves the performance versus the LX generation in most common situations. The report above provides counts for mispredicted branches as well as predicted branch stalls. As with the Xtensa LX case, wider branch instructions in TIE that utilize larger immediates can further mitigate branch delays and are featured with many Xtensa NX configuration DSPs.

The output of the ISS also shows you the percentage of cycles spent in source interlocks. The pipeline interlocks whenever a result takes multiple cycles to compute, and the compiler is not able to schedule meaningful other instructions between the instruction that defines a result and the one that consumes it. Interlocks might be fundamental; some code does not have meaningful parallelism. Some might be a function of the configuration; an 11-stage pipeline in NX may experience more issues with interlocks than the smaller LX pipes.

The Dynamic ISA Profile and Static ISA Analysis tools can be used for Xtensa NX as they are for the Xtensa LX. Non-optimal scheduling might not cause the processor to stall or be filled by NOP operations. For example, these ISA tools are a good way to see frequency of operations, functions, and FLIX formats so you can tune your code accordingly to eliminate bottlenecks.

2.2.5 Compiled Code Quality

In this section we will look at some of the important ways to tune your code to achieve best performance using Xtensa processors. Similar techniques apply to both XCC and XT-CLANG compilers but when deviations exist, those differences are noted.

XCC Optimization

While the ISA profiler and the ISS summary are good tools for obtaining data, nothing can substitute for human intelligence. For important loops, look at the code generated by the compiler and reason about its efficiency. You can either look at the disassembly of the actual object code generated or you can ask the compiler to save the assembly file that it generates before passing it to the assembler. The former is easier to access; for example the Profile

view in Xplorer shows the disassembly along with profile counts. However, the assembly file contains human readable comments that make it easier to understand the generated code.

To see the generated code, compile with either `--save-temps` or `-keep`, or select Keep intermediate compilation files in the Xplorer's Compiler Optimization tab of the Build Properties dialog. As part of the compilation process, the compiler generates a file `name.s` where `name.c` is the original source file name¹. Similar information is annotated in Xplorer by hovering your mouse around the annotation marks in the source code to see vectorization messages, software pipelining feedback, ISA profile, and stack usage.

Every loop in the generated `.s` file is annotated with information to help you understand how the compiler optimized the loop. Most inner loops are software pipelined. Software pipelining is a process where the compiler schedules together operations from multiple loop iterations of a loop. Every software pipelined loop is annotated with a message prefixed with `<swps>`. The following figure is an example from the MP3 application. This example was chosen because it is performance critical, yet simple enough to understand easily.

```
loopgtz a3,.LBB32_ia_mp3_dec_dct64      # [18]

.LBB30_ia_mp3_dec_dct64:                # 0x4a
#<loop> Loop body line 105, nesting depth: 1, iterations: 16
#<swps>
#<swps> 2 cycles per pipeline stage in steady state with unroll=1
#<swps> 2 pipeline stages
#<swps> 3 real ops (excluding nop)
#<swps>
#<swps> min 2 cycles required by resources
#<swps> min 1 cycles required by recurrences
#<swps> min 2 cycles required by resources/recurrence
#<swps> min 3 cycles required for critical path
#<swps> 3 cycles non-loop schedule length
#<swps>
#<freq> BB:30 => BB:30 probability = 0.93750
#<freq> BB:30 => BB:32 probability = 0.06250
.frequency 1.000 15.938
{
    # format ae_format
    ae_lp24x2f.iu    aep0,a2,8          # [0*II+0] id:785
    ae_orp48         aep2,aep2,aep1     # [1*II+0]
}
{
    # format ae_format
    nop
    ae_abssp24s      aep1,aep0          # [0*II+1]
}
```

The first line in the message gives the source position of the loop (making it easy to locate) as well as an estimate of the trip count of the loop. If the trip count is a literal, the compiler will know it exactly. If the count is a variable, the compiler will make an arbitrary guess unless the program has been compiled using the feedback optimization option. With feedback, the compiler will use an average value over all runs.

¹ For compilations using `-ipa`, or interprocedural analysis, the compiler will instead generate a series of files in the directory `binary.ipakeep/` called `1.s`, `2.s`, ... where `binary` is the name of the generated executable.

The next line gives the achieved schedule as well as the unrolling count. When software pipelining, the compiler will try to successively unroll the loop with higher counts until there is no significant performance benefit to unrolling further. When a loop is unrolled, the schedule achieved is the schedule for each unrolled iteration, so divide by the unroll factor to get a schedule in terms of the original loop. Note that for loops with small, literal, trip counts, the compiler might bias towards fully unrolling the loop. In such cases a message might refer to an outer loop in the source that has become an inner loop in the generated code.

For our example, each iteration is scheduled in two cycles. Looking at the code, one of the two instructions contains a NOP operation. At first glance, this appears to be a scheduling problem. On this particular configuration used from the HiFi family of DSPs, the architecture can issue two operations in every VLIW bundle, so one might expect this loop to execute in 1.5 cycles per iteration or equivalently 3 cycles per unrolled iteration.

Looking further, you can see that the compiler believes that this loop requires at least two cycles given the machine resources. That means that if you ignore all dependences and just pack all the operations as tightly as possible, the loop will still require two cycles. As it happens, in this particular architecture, operations `ae_orp48` and `ae_abssp24s` can only be issued in the second slot of a FLIX instruction. Therefore, it is not possible to schedule this loop more efficiently. If the achieved schedule is equal to the resource limit, it is not possible to improve performance by scheduling, that is, by rearranging operations in the loop. Note however, that the limit is a lower bound. Cadence allows irregular VLIW bundling constraints. It is therefore possible that due to an irregular constraint, the actual resource limit of the loop is higher than that computed by the compiler.

The other interesting information is the line labelled cycles required by recurrences. This is an estimate of how many cycles the loop could be scheduled looking only at dependences, assuming infinite resources. If each iteration of the loop can safely be executed in parallel, the recurrence limit will always be 1. Many times a loop that appears to be parallel has a high recurrence limit. This is often caused by potential aliases that the compiler cannot disambiguate. Aliasing is discussed in more detail in [Aliasing](#) on page 55.

Not every inner loop is software pipelined. Some reasons why a loop might not be software pipelined are:

- The loop contains function calls.
- The loop contains branches.
- The loop has a provably small trip count.
- The function is being compiled for space rather than speed².
- The compiler was not able to find a software pipelined schedule.

² A function will be compiled for space if you use the compiler option `-Os` or if you compile with feedback optimization and the function is not frequently executed or if you specify `-Os` in the `-fopt-use` file. For all functions compiled for space, the compiler inserts into the `.s` file a comment `# Optimized for space`

Inner loops that are not software pipelined yet contain no calls or branches are commented with a different formatted message as shown in the following example.

```
#<loop> Loop body line 87, nesting depth: 2, iterations: 2
#<swpf> swp not attempted
#<loop> unrolled 2 times
#<sched>
#<sched> Loop schedule length: 16 cycles (ignoring nested loops)
#<sched>
#<sched>    16 mem refs      (100% of peak)
#<sched>    12 integer ops  ( 75% of peak)
#<sched>    28 instructions (175% of peak)
{
    # format ae_format
    ae_lp24x2f.iu  aep7,a5,8      # [0] id:114
    ae_mulzafp24s.hh.ll  aeq3,aep2,aep3 # [0]
}
{
    # format ae_format
    ae_lp24x2f.iu  aep5,a3,-8     # [1] id:115
    ae_mulzaafp24s.hl.lh  aeq2,aep2,aep3 # [1]
}
...
```

The schedule length gives the total number of cycles to execute one iteration of the loop. Any delay between the end of an iteration and the start of the next is not noted. Inside the square brackets to the right of each instruction is a comment giving the current cycle count of the instruction, assuming a start of 0 on every iteration. Gaps in the numbering shows bubbles in the schedule. Just before the actual instructions is a set of comments giving the number of memory references, ALU operations, and total instructions. Often, you can bound the potential schedule using one of these fields. For example, the particular HiFi architecture used here can only issue one vector load or store per cycle. Because the example schedule issues 16 memory references in 16 cycles, it is not possible to schedule this loop any better. Note that unlike the software pipeliner, this scheduler is much less sophisticated with regards to loop unrolling. Typically, the scheduler will unroll all sufficiently small loops by a factor of two. If you want more significant unrolling, you must unroll in the source code.

To see the effects of higher level optimizations such as vectorization or inlining, compile with the `-clist` option. For a given file `foo.c`, the use of this option will generate a file named `foo.w2c.c` in addition to the normal output files. This file is a C representation of the original file after inlining and loop optimizations. It is meant to be looked at to gain insight into what the compiler does, not to be compiled itself. For compilations using `-ipa`, or interprocedural analysis, the compiler will instead generate a series of files in the directory `binary.ipakeep` called `1.w2c.c`, `2.w2c.c`, ... where `binary` is the name of the generated executable. Note that `-clist` is not supported with C++.

When trying to vectorize your code, the vectorization assistant allows you to graphically browse through your code and see problems that prevent vectorization. To use the assistant, compile your code with `-O3 -LNO:simd`, profile your code and then enable the assistant from the Tools menu.

Consider the following example.

```
void foo(int *a, int *b)
{
    int i;
    for (i=0; i<1024; i++) {
        a[i] = 0;
    }
    for (i=0; i<1024; i++) {
        a[i] += b[i];
    }
}
```

The vectorization assistant can be seen in [Figure 7: Vectorization Assistant](#) on page 40. The first lines, correspond to the second loop which is not vectorizable because the two array references potentially alias each other. The vectorization assistant prints the message "Array base 'a' is aliased with array based 'b' at line 8". The first loop, on line 4, is successfully vectorized.

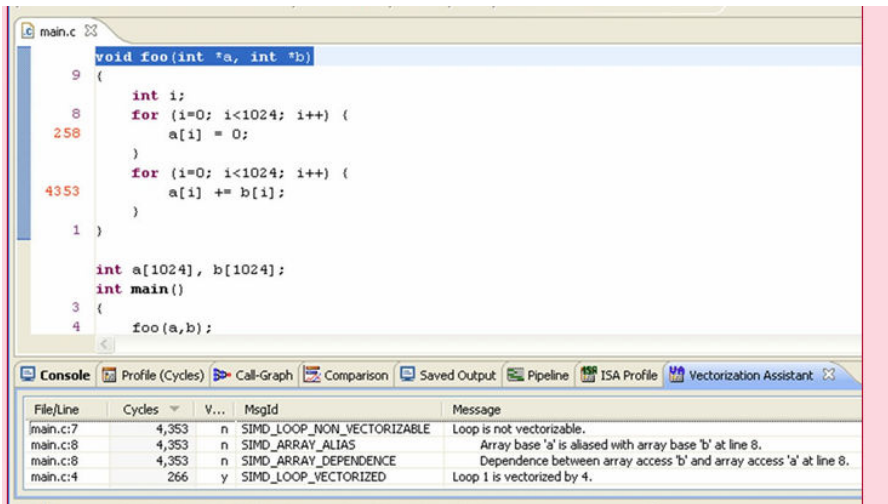


Figure 7: Vectorization Assistant

XT-CLANG Optimization

As with XCC, to see the generated code, compile with `--save-temps` or select Keep intermediate compilation files in the Xplorer's Compiler Optimization tab of the Build Properties dialog. As part of the compilation process, the compiler generates a file `name.s` where `name.c` is the original source file name. The same information is annotated in Xplorer by hovering your mouse around the annotation marks in the source code to see vectorization messages, software pipelining feedback, ISA profile, and stack usage.

Every loop in the generated .s file is annotated with information to help you understand how the compiler optimized the loop as in the XCC case. The following is an example from Xtensa NX configuration from the ConnX family.

```
#loop at line 239
    #64 cycles per pipeline stage in steady state with unroll=1
    #1 pipeline stages
    #49 real ops (excluding nop)
    #17 cycles lower bound required by resources
    #min 64 cycles lower bound required by recurrence
    #min 64 cycles lower bound required by resource/recurrence
    #min 63 cycles required for critical path
    #64 cycles non-loop schedule length
    #trip count: 8
    #register allocated
```

The same information as the XCC compiler is shown in this case, although with slightly different formatting. You can still see the loop heading with line number and cycles achieved in steady state for particular unroll factor. Then there is information on the number of operations used, bounds on resources and recurrences, trip count estimate and registers allocated.

Not every inner loop is software pipelined for similar reasons as in the XCC case (loop conditional code, branches, small trip counts, etc.). Note that the `-clist` option is not supported for XT-CLANG compiler.

When trying to vectorize your code, the vectorization assistant can be used as with the XCC compiler to get insight with feedback messages emitted from the vectorization attempt. To use the assistant, compile your code with `-O3 -LNO:simd`, or alternatively with `-fvectorize`. Profile your code and then enable the assistant from the Tools menu.

Consider a short integer multiply example with output into 32-bit integers and generate the XT-CLANG vectorization messages. The ConnX B10 DSP was used; it can support 8-way 32-bit integer multiply.

```
void test_short_vec(int * __restrict out32, int * in1, int * in2, int N)
{
    int ii;
    for (ii=0; ii<N; ii++) {out32[ii]=in1[ii]*in2[ii];}
}
```

The vectorization assistant can be used again to see the vectorization messages as shown below. Note that you can quickly get the same messages by clicking the green annotation marks in the code perspective of Xplorer that will pop up a window showing similar messages. In this case the loop vectorized 8-way as expected and the message confirms that. If for some reason vectorization was not possible, the message will attempt to provide some reason or hint so the programmer can adjust their code.

```
_restrict out32, int * in1, int * in2, int N) {
```

```
in2[ii];
```

Auto Vectorization

153. vectorization analysis begins with a new loop.

154. Loop is vectorized by 8.

163. vectorization analysis begins with a new loop.

164. Loop is vectorized by 8.

[Show in Vectorization View](#)

XPG View Vectorization Assistant

	Function	Cycles	Instructions	Ve...	Message Type	Message
	test_short_vec	0	0	n	SIMD_PROC_BEGIN	vectorization
2	test_short_vec	0	0	n	SIMD_LOOP_BEGIN	vectorization
2	test_short_vec	0	0	y	SIMD_LOOP_VECTORIZED	Loop is vect

Figure 8: Auto Vectorization Message

3. Performance

Topics:

- [*Controlling Compiler Performance*](#)
- [*General Coding Guidelines*](#)
- [*Floating Point*](#)
- [*C++ Language*](#)
- [*Benchmark Example*](#)

This section looks at specific uses of compiler flags and coding guidelines.

Related Links

[*Performance Tuning Problems*](#) on page 25

This section details the tools and methodologies for maximizing application performance and minimizing code size on Cadence processors.

[*Processor Configuration Options*](#) on page 87

3.1 Controlling Compiler Performance

Whether you use the XCC or XT-CLANG compiler, the compiler has many options to control the amount and types of optimizations it performs. Choosing the right options can significantly increase performance. Compiler options can be controlled with both command line tools and with Xplorer. This section describes multiple compiler options by name. These options can be directly given to the command line for both XCC or XT-CLANG compilers. Inside Xplorer, you can control the compiler options via the Project/Xtensa Project Build Properties menu. Use this menu to set options globally, for a project or for a file inside of a project. The more common optimization flags are check box items in the Optimization tab of the menu. All other options can be entered directly into the Addl tab of the same menu.

By default, the compiler compiles a file with `-O0` (no optimization). The compiler makes almost no effort to optimize the code either for space or performance. The main goals for unoptimized code generation are faster compilation and ease of debugging.

The main optimization flag is the `-On` flag, where `n` can be set between 0 and 3. The flag `-O` by itself is equivalent to `-O2`. For production code, the lowest useful optimization level is `-O2`. The `-O1` level is only rarely useful, for example if you want better debuggability than is possible with `-O2` but cannot use `-O0` because of insufficient target memory.

The `-O3` compiler flag enables additional optimizations that are mostly (but not exclusively) useful for DSP-like code. While the `-O2` flag will almost always result in better code than `-O1`, the `-O3` will usually result in faster code than `-O2` but not always. Some code will speed up by a factor of two while other will slow down a little. It is not possible to predict in advance. Most programs will be larger with `-O3` than `-O2`.

The use of `-Os` flag tells the compiler to optimize for space rather than speed. For XT-CLANG compiler if the flag `-Os` is used with either `-O2` or `-O3`, only `-Os` will be considered and the other option will be ignored. In comparison to `-O2`, `-Os` generated code will on average be 10% slower, but 15% smaller. In many scenarios, you might want to compile your important functions for speed and the rest for space. When using feedback directed compilation without `-Os`, the compiler automatically compiles less important functions for space. In addition, compile option `-Oz` is a more aggressive version of `-Os`.

The `-g` option instructs the compiler to generate symbolic debugging information when compiling. It can be used together with other optimization flags without adversely affecting performance. This flag is required for symbolic debugging and for line by line profiling.

3.1.1 Optimizing Functions Individually

Compiler flags allow the user to control optimization but only at the file level. Each flag applies to all of the functions in a file. It is often useful to compile different functions in the same file using different compiler flags. For the XCC compiler, this can be done either using

attributes or using an external editable text file, known as an optimization file, using the *-fopt-gen* and *-fopt-use* flags.

The *-fopt-use* flag may be used to specify function optimization levels by placing a list of one line entries in a text file and supplying that file name through the *-fopt-use* flag. The flag usage is as follows: *-fopt-use=<optimization_file>*, where the *<optimization_file>* is a text file with a list of lines, each with the general format:

[<directory_path>]<filename>:<procedure_name>: <optimization_string>

The optional *<directory_path>* may be needed to disambiguate file name collisions. Lines beginning with # are ignored as comments. The longest line allowed is 1024 characters. The format for the *<optimization_string>* for function-level optimization is as follows: *[-O{0,1,2,3}][-Os]* to indicate compiling a certain function at levels 0 through 3, and/or to optimize for space. The optimization levels specified in the file will override any command line flags except for the *-O0* flag. The command line *-O0* flag overrides any file specifications, allowing easier debugging.

For example, create a text file name *foo.opt*, with the following contents:

```
test.c:test_foo: -O1
test.c:test_bar: -O3 -Os
main.c:main_foo: -O0
```

Then compiling with command line: *xt-clang -O2 test.c main.c -fopt-use=foo.opt* would result in the following: *test_foo()* in *test.c* compiled at *-O1*, *test_bar()* in *test.c* compiled at *-O3* and optimized for space, *main_foo()* in *main.c* compiled at *-O0* and any other function in *test.c* and *main.c* compiled at *-O2*, as specified on the command line.

Using the compiler flag *-fopt-gen* at both compile and link time will cause the compiler to create an output file named *executable.opt* mentioning every function in the application in a format that is suitable for use by *-fopt-use*. For most functions, the optimization will be set to whatever was given in the command line. However, if feedback compilation is used, the compiler will automatically compile certain functions for space. By generating and then keeping the generated file, the user can take advantage of the decisions made by the feedback optimization without having to keep compiling the application using feedback. The use of the *-fopt-use* flag is meant to be robust to changes in the application. If a function is later deleted, the compiler will ignore its entry in the file. If a function is added and no entry is added to the file, the function will be compiled with whatever flags are given on the command line.

C++ functions must be specified using their mangled names. By starting with the *-fopt-gen* flag, the file will contain the correct mangled name of every function. The standard utility *c++filt* can be used to demangle the names in the file to get their unmangled C++ name.

Note that the XT-CLANG compiler does not yet support per function option compilation.

Rather than use a separate file, one can also use the attribute feature in the source file to set the optimization level for the attributed function to `opt_level`, where `opt_level` is one of `-O0`, `-O1`, `-O2` or `-O3`, possibly paired with `-Os`. This attribute instructs the compiler to compile the function at the specified optimization level regardless of the command-line optimization level. This feature is fully supported in XCC. For XT-CLANG compiler only `-O0` and `-Os` options are supported. For example, the following attribute instructs the compiler to compile the function `func0()` for space rather than speed.

```
__attribute__((optimize ("-Os"))) void func0() {
    int i;
    for(i = 0; i < 100; i++) {
        a[i] = b[i] + c[i];
    }
}
```

3.1.2 Attributes

Both XCC and XT-CLANG compilers support attributes for giving the compiler information beyond what is expressible in standard C or C++. Attributes are used similarly to pragmas in other compilers and are often used to control memory placement, alias analysis and inlining, for example. The XCC and XT-CLANG compilers support most of the common GCC and LVM attributes, and give warning messages when encountering unsupported ones. See the *Xtensa C and C++ Compiler User's Guide* and *Xtensa XT-CLANG Compiler User's Guide* for more details on what is supported. Following is a table for quick reference of common attributes and how to declare for each compiler.

Table 2: Common Attributes

Description	XCC	XT-CLANG
Aligning hint	<code>__attribute__((aligned (n)))</code>	<code>__attribute__((assume_aligned (n)))</code>
Control inline	<code>__attribute__((always_inline))</code> <code>__attribute__((noinline))</code>	<code>__attribute__((always_inline))</code> <code>__attribute__((noinline))</code>
Data placement	<code>__attribute__((section ("dram0.data")))</code>	<code>__attribute__((section ("dram0.data")))</code>
Alias hint	<code>__restrict</code>	<code>__restrict</code>

3.1.3 Interprocedural Analysis

One of the distinguishing features of the XT-CLANG or XCC compiler is its ability to perform interprocedural analysis (IPA) and optimization. While most traditional compiler optimizations are done within a single function, interprocedural optimizations are applied to the whole program. Various compiler analyses yield more precise information when performed across the entire program, and this in turn enables additional optimizations that may not be possible within the separate compilation model.

Interprocedural Optimizations Implemented by XCC

- Improved function inlining, with heuristics based on the analysis of the call graph for the entire program.
- Constant propagation for function parameters that are always passed the same constant value.
- Dead function and variable elimination to reduce the program size (especially when used in conjunction with the `-Os` option).
- Identification of global variables that are initialized to constant values, and never modified.
- More precise alias analysis based on the whole program view, which often may eliminate the need to use more restrictive memory aliasing models.

To enable interprocedural analysis and optimization, you must use the `-ipa` (or `-IPA`) command-line option both when generating object (`.o`) files and when linking them. When this option is in effect, the `.o` files produced by the compiler will not be the normal object files; instead, they will contain information about the original code, summarized in a form that is suitable for interprocedural analysis. During the link step, rather than invoking the standard linker (`xt-ld`), the XCC driver invokes the interprocedural module, which digests the summary `.o` files and performs various analyses and transformations.

This module then generates intermediate files and writes them into a temporary directory, `binary.ipakeep`, whose contents can be saved with `-keep`. In the final step, it invokes the compiler to transform the intermediate files into real object files, and then the standard linker to create the final output.

Because the interprocedural optimizer mimics the traditional steps of compiling and linking, most makefiles for normal executables will continue to work with a simple addition of `-ipa` to the command-line options. However, you should be aware of the following caveats:

- The `-ipa` option should be used for both compiling and linking. Although the interprocedural module will accept normal `.o` object files (produced without `-ipa`) and generate correct code, it cannot derive information useful for its analyses from such files. An attempt to link interprocedural summary `.o` files without the `-ipa` option on the link line will result in an error message produced by the linker.
- You must use the XCC compiler driver (`xt-xcc` or `xt-xc++`) during the link step with IPA. Using the standard linker (`xt-ld`) will result in an error message.
- The linking step with the interprocedural module hides the entire back-end compilation and optimization, in addition to the processing of summary `.o` files. Therefore, re-linking even after changing just a single `.o` file takes much more time than without IPA.
- The interprocedural module can inline functions across file boundaries. If different command line options are used for different files compiled with IPA, which options are used for any particular function is not defined. Therefore, it is recommended (although not required) that you use the same command-line options (especially the optimization level) for all the files compiled with IPA.
- Because of the dramatic extent of code changes performed during interprocedural optimization, using `-g` in conjunction with `-ipa` will produce extremely limited debugging

information. This information will allow you to do line profiling (allow a profiler to show how much time was spent in different source lines) as well as set breakpoints and step through the code in a debugger. However, examining source-level data in a debugger will not be possible.

- The `-ipa` option has no effect if it is used in conjunction with a command-line option that terminates the compilation process before producing object files (`-E` or `-S`).

To use interprocedural analysis when generating a final executable, compile the `.o` files with the `-ipa` option, and use the `-ipa` option on the link line.

To use interprocedural analysis on archives or `.a` files, compile the `.o` files with the `-ipa` option, and use the `-ipa` option on the link line.

Interprocedural analysis can be used in conjunction with the other optimization flags. For example, you can use `-ipa` together with `-O2` or `-O3` or `-Os`. When compiled for speed, IPA tends to inline heavily and thus increase code size. When compiled for space, IPA is capable of deleting unused functions and variables and thus is sometimes able to significantly reduce memory requirements. When compiled with feedback, IPA will automate the trade-off between speed and space and will tend to give both better speed and space characteristics.

Interprocedural Optimizations Implemented by XT-CLANG

Interprocedural analysis and optimization are implemented very differently in XT-CLANG, enabled with `-flto` for Link Time Optimization; therefore, performance differences between the two compilers are to be expected. However the benefits and motivation are similar to those mentioned in the previous section on XCC. The `-flto` option should be used for both compiling and linking, and the link step should be done with `xt-clang/xt-clang++`, and not directly with `xt-ld` as in the XCC case. Options `-Os`, `-Oz`, `-LNO:simd` and some others in case of LTO are remembered in the intermediate representation in the object file as the function attributes. They may be different for different object files if needed. XT-CLANG has also an option to run a "light-weight" LTO in terms of compile time, ThinLTO. It is enabled with option `-flto=thin` (instead of regular `-flto`). Regular (full) LTO is performed by merging all input into a single module, while ThinLTO only merges compact summaries which allows cross-module function importing between modules, but they are still built separately. In order to use ThinLTO, the `-flto=thin` option should be used for both compiling and linking with `xt-clang/xt-clang++`.

Building Libraries with IPA

The following discussion describes the XCC compiler method. XT-CLANG supports similar options `-flto-lib` and `-flto-entry`.

An archive built using the archiver `xt-ar` on object files compiled with IPA (or LTO) is not truly compiled until link time. The library itself, before link time, contains a representation of the preprocessed source files (compiler intermediate representation in case of LTO) in the library. Therefore, IPA (or LTO) is required to be used at link time. This is useful because with IPA (or LTO) both the application and the functions it calls from the library can be optimized together.

For example, IPA (or LTO) can inline a function from the library into the main application, an optimization that is not possible with non-IPA compiled libraries.

However, sometimes it is useful to build a normal compiled library using IPA (or LTO). Such libraries can be distributed and then be linked into an application without requiring the final link to use IPA. When building the library, the compiler might inline functions if both the caller and the callee are within the library, but functions in the library will never be inlined into the application. Applications linked against these libraries can still use IPA to optimize the application interprocedurally, but the library will not be recompiled.

To safely optimize a program, IPA needs to know which functions access the global state. When building an executable, IPA sees all the object files, whether real or IPA objects. Therefore, IPA is able to automatically compute the information it needs in order to safely perform optimizations. When building a normal, compiled, library, IPA has no way of knowing what variables and functions in the library might be referenced by an application that will be linked to that library. For example, IPA can delete unused functions. But a function in a library might be designed to be called from a library's client, and not within the library itself. That function will appear to be unused and normally IPA would delete it. Therefore, when building a compiled library, you must explicitly tell IPA which variables and functions are externally accessible.

To build a normal, compiled, library with IPA, compile the source files with `-ipa (-fcto)` added to the compilation flags. Linking the library requires two flags: `-ipalib (-fctolib)` and `-ipaentry=symbol_name (-fctentry=symbol_name)` repeated for each symbol (function or variable) that is externally accessible. For optimization purposes, IPA will assume that all the symbols marked with an `-ipaentry=symbol_name` may be accessed arbitrarily outside the library and will also assume that no additional global symbols are accessed.

3.1.4 Inlining Functions

The XCC and XT-CLANG compilers inline functions; that is, replace a function call with the actual body of the function. Inlining significantly improves performance for small functions.

Inlining Functions by XCC

XCC will inline both with and without IPA. Without IPA, XCC can only inline functions that are defined in the same file as the caller (either directly or via a header file that is included in the calling file). Even then, by default without IPA XCC will only inline static or inline functions. This is done to improve code size because non-static files must still be emitted even if they are inlined because they might also get called from other files. If you want to inline non-static functions, compile with the `-INLINE:preemptible` flag. With IPA, by default the compiler will potentially inline any function no matter how it is defined and no matter from where it is called.

The compiler uses its own heuristics to decide which functions to inline. Even if a function is marked as inline, the compiler might still decide not to inline the function. Deciding on the benefits of inlining is very complicated, and the compiler does not always make the right

decision. The compiler gives you a lot of freedom if you want to control its inlining heuristics. You can request that a specific function always or never be inlined using the flags -`INLINE:must=function` or `-INLINE:never=function`. A function marked never to be inlined will not be inlined. A function marked must to be inlined will be inlined if possible. Certain function calls, such as recursive calls, can never be inlined. Similar behavior can be achieved via attributes in the code rather than via compiler options. As shown in the following two examples, a function attributed as `always_inline` will be inlined if possible while a function attributed as `noinline` will never be inlined.

```
int foo () __attribute__ ((always_inline));
static int foo2 () __attribute__ ((noinline));
```

Inlining can also be controlled more globally. The option `-INLINE:requested` causes the compiler to inline all functions marked as inline, if possible. The option -`INLINE:requested_only` inlines all functions marked as `inline` if possible and inlines no other functions.

Inlining Functions by XT-CLANG

By default XT-CLANG is much more aggressive in inlining fuctions. You can revert back to the C89 standard methods with option `-fgnu89-inline`. XT-CLANG has less controllability for inline options. The following table shows the comparison between the two compilers on common inline options.

Table 3: Inline Option Comparison

Option	XCC	XT-CLANG
Ignores the inline specifier and does not inline any functions	<code>-fno-inline</code>	
Disable inlining of functions based on heuristics	<code>-fno-inline-functions</code>	<code>-fno-inline-functions</code>
Inline functions that are (explicitly or implicitly) marked inline	<code>-INLINE:requested_only</code>	<code>-INLINE:requested_only</code> <code>-finline-hint-functions</code>

3.1.5 Using Profiling Feedback

Various compiler optimizations may benefit from profiling information; that is, how frequently different regions of code are executed. As one example, when the Xtensa processor executes a conditional branch instruction, it incurs additional overhead when the branch is taken compared to falling through to the next instruction. Therefore, it is desirable to have as many branches be fall-through as possible, which the compiler can achieve by reordering the code and changing the branch directions. As another example, the compiler should not unroll

a loop that in practice often only executes for one iteration, and the inliner should, in general, only inline functions that are frequently called.

XCC Compiler

XCC has three mechanisms to estimate the frequency of different regions of code: heuristics, pragmas and automatic feedback from profiling information.

In the absence of other means, XCC uses heuristics for guessing branch directions. For example, XCC assumes that loops execute multiple iterations. Of course, heuristics are just guesses, and XCC can do a much better job with more accurate information. XCC has a mechanism to feed back information taken from an actual run back into the compilation process, using the following three-step process:

1. In the first step, set the top level **FBmode** switch to either **SW** or **HW** depending on whether you intend to generate feedback data in simulation or on a hardware target. Select a target whose name ends in **_Feedback**. Alternatively, from the command line invoke the compiler with one of the following options:

```
xt-xcc ... -fb_create filename ....
```

The compiler instruments the code to count the frequency of all branches. By default, the command-line compiler uses 32-bit counters (Xplorer always uses 64-bit counters). If your code is sufficiently long running so that any particular region is executed more than $2^{31} - 1$ times, the counters will overflow and generate inaccurate counts. This will not cause your program to execute incorrectly, but it will degrade optimization. For such long-running programs, invoke the compiler including the following option to use 64-bit counters.

```
-fb_create_64 filename
```

Note that for this first step, you must both compile and link your application with this flag. Also note that the compiler instrumentation uses standard library functions to do memory allocation and file I/O. If you redefine any of the library functions used by the instrumentation library, you cannot compile any file containing those redefined functions using `-fb_create`. XCC gives an error message in such situations. If you redefine any library function in a way that redefines its externally visible behavior, even if you do not compile the function with `-fb_create`, you may get unpredictable results in the instrumented version.

2. In the second step, run your application with a representative input set. In the first step, Xplorer automatically created a target named *target_Feedback* where *target* was the name or the target used when setting the optimization flags. Use that *target*. You may run the application multiple times with different representative sets. Each run creates a new profiling file prefixed by *filename*. The program can be run on any system that supports basic file I/O, including the simulator. This run will take significantly longer than a normal run, often several factors longer. The input sets do not have to be the same as the ones used in production (often, shorter running input sets are used). However, the closer

the run is to the production run, the more accurate the information. If a particular run only invokes some subset of the common modes expected in production, it is possible and desirable to execute multiple runs.

If your target hardware does not support a file system, in the first step, set the mode to **HW**. From the command line, use the following option:

```
-fb_create_HW filename
```

The compiler instruments the code as in the `-fb_create_64` case, but also includes software to allow the data to be retrieved on real hardware automatically through the debugger. Code compiled and linked with this option must be executed on the hardware with the debugger attached, either through Xplorer or through xt-gdb using an OCD target. Once the file is retrieved, it can be used identically as files produced with the other options. See the *Xtensa Software Development Toolkit User's Guide* for more information.

3. In the third step, switch the target back to the original target (making sure that FBmode is still set) or from the command line reinvoke the compiler including the following flag:

```
xt-xcc -fb_opt filename
```

The compiler uses the profiles generated in all the files prefixed by *filename*. Multiple profiles are averaged together, weighted by their execution time. This second invocation of the compiler must use the same sources and similar flags as the first invocation using `-fb_create filename`. In particular, you must either use `-ipa` in both compilations, or neither. Xplorer will ensure that *target* and *target_Feedback* will remain in sync. If you change the sources, you must regenerate your profiling files by going back to the first step. Command-line users must be sure to delete old profile files; Xplorer will do this automatically. If you do not delete the old profile files, the compiler shows an error message saying that the profiling files no longer match.

Note that the profiling file is created in the same directory in which the application is run. If you compile and run it in separate directories, you must either move the profiling files, or use a full path to the run directory when performing the third step.

Using automatic feedback can significantly improve run-time performance, but it makes an even larger impact on code size, because it enables XCC to automatically decide which functions to compile for speed and which functions to compile for size. When using automatic feedback, XCC compiles every routine that takes at least one percent of the execution time for speed and every other routine for size by default. This ratio can be adjusted using the following flag:

```
xt-xcc -OPT:space_opt=n
```

Using this flag, XCC optimizes a function for space whenever that function takes less than $(n/10)\%$ of the total execution time. The default value of n is 10, meaning that XCC optimizes for space any function that takes less than 1% of the total execution time.

Note that to take advantage of this feature of feedback, your configuration must have at least one timer or you must compile with IPA.

When you cannot use automatic profile information, or when you know that a branch is almost always taken (or not taken) regardless of feedback information, XCC provides two pragmas:

```
#pragma frequency_hint NEVER
#pragma frequency_hint FREQUENT
```

When placed right after the conditional test of an `if` statement (at the beginning of the `then` block), these directives indicate that the conditional branch is almost never, or very frequently taken. In the following example,

```
int a_lt_b(int a, int b)
{
    if (a < b) {
        #pragma frequency_hint NEVER
        return 1;
    }
    return 0;
}
```

the `frequency_hint` directive indicates that the branch is almost never taken, and XCC produces the following assembly code at the `-O2` optimization level:

```
blta2,a3,.LABEL
movi.na2,0
retw.n
.LABEL:
movi.na2,1
retw.n
```

If the frequency hint is changed from `NEVER` to `FREQUENT`, the generated code changes to:

```
bgea2,a3,.LABEL
movi.na2,1
retw.n
.LABEL:
movi.na2,0
retw.n
```

XT-CLANG Compiler

Just like the XCC case, the XT-CLANG compiler uses heuristics, pragmas, and profile information to determine the frequency of various sections of code to optimize schedule. For

example, users may want to use `__builtin_expect()` to specify what is the expected value for the condition expression. The user can say "if (`__builtin_expect(x<y, 0)`)" to declare a condition's expected value.

The feedback profile information is done in different way and discussed below.

Profile information that is later feedback to the compiler enables better optimization. For example, knowing that a branch is taken very frequently helps the compiler make better decisions when ordering basic blocks.

Step 1. Compiler with profiling collection.

You must pass `-fprofile-instr-generate` option, that is:

```
xt-clang++ -O2 -fprofile-instr-generate code.cc -o code
```

Step 2. Run the instrumented executable with inputs that reflect the typical usage (ISS only).

By default, the profile data will be written to a `default.profraw` file in the current directory. You can override that default by using option `-fprofile-instr-generate="/path/to/profile/profile.profraw%m"`. Note that due to ISS limitations a new directory cannot be created. Therefore `"/path/to/profile"` must be existing path.

Note on `"%m"` modifier. By default, profile file will be overwritten after each program run. Appending `"%m"` modifier to the file path will prevent overwriting by enabling profile merging in PGO runtime. This step must be done on ISS.

Step 3. Convert the "raw" profile format to the input expected by XT-CLANG.

Use the `merge` command of the `xt-profdata` tool to do this.

```
$ xt-profdata merge -sparse -o profile.profdata profile.profraw
```

The `profile.profdata` file will be generated.

Step 4. Build the code again using collected profile data.

Pass `-fprofile-instr-use=` option to specify the profile file.

```
xt-clang++ -O2 -fprofile-instr-use=profile.profdata code.cc -o code
```

You can repeat this step as often as you like without regenerating the profile. As you make changes to your code, XT-CLANG may no longer be able to use the profile data. It will issue a warning when this happens.

3.1.6 Aliasing

Consider the code in the following example. You might assume that the compiler will generate code that loads `*a` into a register before the loop and contains an inner loop that loads `b[i]` into a register and adds it into the register containing `*a` in every iteration.

```
void foo(int *a, int *b)
{
    int i;
    for (i=0; i<100; i++) {
        *a += b[i];
    }
}
```

In fact, you will find that the compiler generates code that stores `*a` into memory on every iteration. The reason is that `a` and `b` might be aliased; `*a` might be one element in the `b` array. While it is very unlikely in this example that the variables will be aliased, the compiler cannot be sure.

There are several techniques to help the compiler optimize better in the presence of aliasing. One technique is to compile using `-ipa` option for XCC compiler or `-fllto` for XT-CLANG compiler. This way, the compiler sees the entire program and might be able to figure out in the above example that `*a` cannot be an element of the `b` array. Using this technique has some nice advantages; it is easy and it is safe. However, the compiler is not always able to figure out what is obvious to the programmer. Consider the next example. Sometimes the first parameter to `foo` is the global `a` and sometimes it is `b`, similarly for the second parameter. Unless the function happens to be inlined, the compiler is not smart enough to realize that the first parameter is only `a` when the second parameter is `b`, and therefore the compiler conservatively assumes the two parameters are aliased.

```
int a[100], b[100];
void foo(int *a, int *b)
{
    int i;
    for (i=0; i<100; i++) {
        *a += b[i];
    }
}
int bar()
{
    foo(a,b);
    foo(b,a);
}
```

Alternatively, you can use the global variables directly instead of parameters. The compiler knows that two globals cannot alias with each other. Of course, globals cannot be used inside of functions that access different data in different invocations and the overuse of globals does not promote good software engineering practices.

Parameters or local pointers scoped at the function level can have their type qualified with the `__restrict` type qualifier as shown in the next figure. Pointers with the `__restrict` type qualifier are assumed by the compiler to not alias any other pointers or globals for the entire dynamic duration of the function. The type qualifier is easy to use and is effective. However, you must be very careful. If two restrict pointers do in fact refer to overlapping memory locations, the compiler may generate incorrect code. Users frequently use this feature when they are not sure if pointers are aliased, and then just check if the program still generates correct results. Unfortunately, incorrect usage of restrict may or may not lead to incorrect results. So, the program may generate correct results when the qualifier is used but might start generating incorrect results after random changes are made to the code.

```
void foo(int * __restrict a, int * __restrict b)
{
    int i;
    for (i=0; i<100; i++) {
        *a += b[i];
    }
}
```

The restrict type qualifier also interacts with inlining. Consider the next example, where the compiler has been told that the portion of array `a` accessed in functions `barney` or `wilma` does not overlap with the portion of array `b` accessed in the same function. The user has said nothing about whether the portion of array `a` accessed in `barney` overlaps with the portion of array `b` accessed in `wilma`. After inlining, the compiler cannot distinguish between array references that originally came from `barney` from those that originally came from `wilma`. Therefore, the compiler cannot inline the two functions and still keep the restrict qualifier. Given the choice between preventing the inlining and dropping the restrict qualifier, the compiler chooses to prevent inlining.

```
void barney(int * __restrict a, int * __restrict b);
void wilma (int * __restrict a, int * __restrict b);
void fred(...)
{
    barney(a, b);
    wilma(a, b);
}
```

Finally, the XCC compiler has an option, `-OPT:alias=value`, to globally control how the compiler handles alias analysis. Value can be one of `any`, `typed`, `restrict` and `disjoint`. **Note:** The `-OPT` options do not exist with XT-CLANG compiler. Instead you can use the `-fstrict-aliasing/-fno-strict-aliasing` options described below. The option `-OPT:alias=any` (or equivalently `-fno-strict-aliasing`) tells the compiler to make no assumption about aliasing. Unless the compiler can prove that two variables or pointers do not overlap, the compiler will assume that they do. The option `-OPT:alias=typed` (or equivalently `-fstrict-aliasing`) tells the compiler to assume that pointers to different types do not overlap. So for example, a `float *` pointer will not point to the same location as an `int *` pointer. Similarly, a pointer to a particular field of a structure will not overlap to a

different field of any structure even if the underlying types of the fields are the same. With this rule, there is a special exception for variables of type `char *`. The compiler will conservatively assume that such pointers can alias with any other. These type based rules are requirements of the C and C++ language. A program that casts an `int *` pointer to a `float *` pointer, and then uses both pointers, is not a legal C or C++ program. Therefore, `-OPT:alias=typed` is the default behavior of the compiler and you must explicitly set `-OPT:alias=any` if your code violates the standard.

The option `-OPT:alias=restrict` instructs the compiler to treat every pointer as if it had the type qualifier `__restrict`. This option should only be used in very limited circumstances where you are sure that there are no aliases. Frequently, this option is useful as a quick testing mechanism to see if the compiler is being limited by aliasing in a particular loop. If using this flag does not improve performance, there is no point in trying any of the more conservative methods.

Even the option `-OPT:alias=restrict` does not eliminate all aliases since it tells the compiler that two pointers are not aliased but says nothing about the indirection of two pointers. Consider the following example. Using the option `-OPT:alias=restrict` tells the compiler that `*a` cannot equal `*b` any place in the function but it does not say that `**a` cannot equal `**b`. The flag `-OPT:alias=disjoint` is a stronger flag; it tells the compiler that arbitrary indirections off of named pointers do not alias with other pointers or with other named variables.

```
void foo(int **a, int **b)
{
    int i;
    for (i=0; i<100; i++) {
        **a += **b++;
    }
}
```

3.1.7 Loop Pragmas

XCC and XT-CLANG compilers support a set of pragmas to guide the compiler in optimizing loops. These pragmas impact the vectorizer, but also other portions of the compiler. A loop pragma must be placed immediately preceding the loop to which it applies.

Common Pragmas

```
#pragma no_unroll (XCC or XT-CLANG)
#pragma nounroll (XT-CLANG)
```

This pragma disables unrolling of the loop it immediately precedes.

```
#pragma loop_count min=<level>, max=<level>, factor=<level>, avg=<level>
```

This pragma asserts the minimum trip count ³ (min), the maximum trip count (max), and the trip count's even divisibility (factor) of the loop it immediately precedes. The minimum and factor values must be exactly correct as the compiler may use this information for optimizations such as omitting unrolling remainder iterations. Incorrect values for these parameters could result in incorrect code.

The use of the min value allows the compiler to improve performance when software pipelining loops. When software pipelining, the first few and the last few iterations are split up into loop prologues and epilogues. In order to handle the situation where the trip count of the loop is smaller than the number of prologues or epilogues, the compiler must schedule each one separately and test for an early exit after each one. The use of the pragma avoids this overhead. The use of the factor value allows the compiler to avoid generating cleanup loops when vectorizing or unrolling loops. It will not typically have a major performance impact but might substantially reduce code size.

For either compiler, you can disable the use of this pragma with the `-fno-pragma-loop-count` command-line option. Additionally, the average value (avg) provides a way to tell the compiler an estimate of the average trip count for this loop. The compiler will try to use this in heuristics that require an estimate of the amount of work done each time the loop is entered.

Other useful pragmas are `#pragma concurrent`, which allows the compiler to assume loop iterations are independent and proceed with vectorization, and `#pragma simd_if_convert`, which allows for predicated vectorization by unconditionally executing all the paths in control flow conditional statements and results are committed conditionally with appropriate operations.

XT-CLANG Compiler

XT-CLANG supports most of the XCC pragmas and some additional ones, some of which are listed in the following table. They refer to techniques to control loop vectorization and loop unrolling to various degrees. These pragmas are inserted before the loop of interest to affect the code in the loop.

Table 4: Additional XT-CLANG Pragmas

<code>#pragma clang loop vectorize(enable)</code>	EnableVectorization
<code>#pragma clang loop vectorize_width(n)</code>	Set vectorization width
<code>#pragma clang loop unroll(enable)</code>	Unroll loop fully or to some limit
<code>#pragma clang loop unroll(full)</code>	Unroll loop if count known at compile time
<code>#pragma clang loop unroll_count(n)</code>	Set unrolling factor

³ The trip count is the number of times the loop body is executed.

3.1.8 SIMD Vectorization

Many coprocessors, including the ConnX, HiFi, Fusion, FPX, and Vision family of cores contain SIMD (or vector) instructions, meaning, instructions that perform the same operation on multiple pieces of data, each piece occupying a slice of a register file or memory location. SIMD offers a great potential for performance improvement. The user can manually take advantage of SIMD using explicitly SIMD intrinsics or operator overloading on explicitly SIMD data types. The compiler is also able to vectorize code, automatically convert scalar code into vector code. This subsection is mainly concerned with automatic vectorization. Vectorization can be difficult to utilize effectively. To utilize vector techniques, three general conditions need to be met:

- First, fundamentally the algorithm being implemented must be performing the same independent operations on multiple pieces of data. If the algorithm is not fundamentally amenable to SIMD, it will not vectorize.
- In the case of XCC compiler, the multiple pieces of data must be contiguous in memory. Consider a simple image processing example operating on RGB data. Imagine that a particular algorithm is manipulating only the red data. RGB data is typically stored in one of two ways; separate arrays for each component or a single, interleaved array. If the algorithm is only manipulating the red data, the red data must be stored in its own array. The XCC compiler is not going to deinterleave the storage for you. In the case of XT-CLANG, the compiler can handle interleaving of data up to 4, for example reading from indexes $4*i$, with i being the counting index.
- Third, the compiler must be able analyze the code and prove that the first two conditions hold. The remainder of this subsection discusses this third condition.

Vectorization is invoked in one of two ways:

1. Vectorization can be invoked as a check box in the Optimization tab of Xplorer's Build Properties menu when developing software for a configuration that supports vector instructions.
2. Vectorization can be invoked on the command line by using the `-O3 -LNO:simd` command-line option. In XT-CLANG you can also use `-fvectorize`.

[Table 5: Vectorization Options](#) on page 59 summarizes the options that control the vectorizer for each compiler. Note that some options are not available in one or the other compiler. The remainder of this subsection describes the use of the vectorizer in more detail.

Table 5: Vectorization Options

Feature	XCC	CLANG
Autovectorization	<code>-LNO:simd</code>	<code>-fvectorize -LNO:simd</code>
Alignment	<code>-LNO:aligned_pointers=on</code>	

Feature	XCC	CLANG
	- LNO:aligned_formal_pointers =on	
Conditional vectorization and speculation	-LNO:simd_agg_if_conv -TENV:X=n	-LNO:simd_agg_if_conv

Automatic vectorization, and the `-LNO` options that control it are only available in conjunction with the `-O3` option. At this optimization level, the compiler performs loop-nest dependence analysis that is used to evaluate validity and profitability of vectorization transformations.

There are several possible approaches to using vector instructions within your C or C++ program:

1. Do not modify your code at all, and simply use the automatic vectorization feature in XCC or XT-CLANG compiler. This is the easiest method, but it may not take full advantage of all hardware capabilities. Note that the degree of vectorization varies between the two compilers.
2. Modify your code to meet one of the constrained memory models described in [Aliasing](#) on page 55. Depending on the source code, these modifications may be relatively easy or very difficult. This task is more difficult than not modifying your code at all, but can result in dramatically improved performance.
3. Rewrite the code partially or completely using explicit vector DSP types along with operator overloading on those types.
4. Rewrite the code partially or completely using Xtensa processor's TIE intrinsic functions. This method is the most time-consuming (although it is not very difficult), but allows you total control and has the most potential for improving the performance of time-critical portions of your application.
5. Language extensions based vector approach as defined in <https://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors>. You can use such method for XT-CLANG only and for standard C types like float and int (not fixed point Q format types supported in some DSPs).

Combinations of these methods can be applied appropriately to the various algorithms.

3.1.8.1 Viewing the Results of Vectorizing Transformations

When writing and debugging SIMD code, it is often useful to see how the compiler has optimized the code you originally wrote.

3.1.8.1.1 For XCC

As described in [Compiled Code Quality](#) on page 36, you can look at the `.s` assembly file that the compiler generates if the `-save-temps` or `-keep` command-line option is in effect. But a higher, C-level view may be much easier to understand.

To see how the vectorizer has transformed your code, use the `-clist` option on the command line or select the *Produce a .w2c.c file* option in the Optimization tab of Xplorer's *Project/Xtensa Project Build Properties* menu. XCC produces a C language translation of the optimized version of your code in files `<filename>.w2c.c` and `<filename>.w2c.h`. (If you use `-clist` in conjunction with `-ipa`, the translated files are named `1.w2c.c`, `2.w2c.c`, and so on, corresponding to the intermediate files created by the interprocedural module in the `<executable_name>.ipakeep` directory.) These files contain readable code that you can examine to find out which parts of your program have been vectorized. Although the translated code is usually, but not always, compilable, this is not the intended use of this option - there is no guarantee that the generated C code is semantically equivalent to the original source code.

Consider the following example:

```
int a[100], b[100], c[100];
int main()
{
    int i;
    for (i=0; i<100; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Compile the example targeting the Fusion G3 coprocessor using

```
xt-xcc -O3 -LNO:simd -clist main.c
```

As a by-product of compiling `main.c`, the compiler generates two files: `main.w2c` and `main.w2c.h`. A slightly edited version of `main.w2c.c` follows.

```
#include "main.w2c.h"
__INT32 main()
{
    xb_vecMx32 V_00;
    xb_vecMx32 V_;
    xb_vecMx32 V_0;
    xb_vecMx32 V_4;
    __INT32 reg2;
    __INT32 i;

    for(i = 0; i <= 99; i = i + 4)
    {
        V_00 = *(xb_vecMx32 *)(&b[i]);
        V_ = *(xb_vecMx32 *)(&c[i]);
        V_0 = PDX_ADD_MX32(V_00, V_);
        V_4 = V_0;
        *(xb_vecMx32 *)(&a[i]) = V_4;
    }
    return reg2;
} /* main */
```

Translating the compiler's internal representation into C creates the `.w2c.c` files. As the code in these files is readable, you can see what portions of the original program were vectorized. You can use the information to manually vectorize the source or to aid its automatic vectorization. The `.w2c.c` files represent the stage of the compilation process immediately after vectorization. Further optimizations, including loop unrolling, happen at later states in the compilation process and are therefore not seen in the `.w2c.c` files.



Note: `-clist` is not currently available for C++ programs.

3.1.8.1.2 For XT-CLANG

The XT-CLANG compiler does not support the `-clist` option to see intermediate compiler transformation files of your code. However, the `.s` file with the assembly information can still be generated normally the same way as for XCC by inserting the `-S` or `-save-temps` compile options.

3.1.8.2 Aligning Data for Vectorization

The Xtensa architecture requires all DSP memory references to be aligned. Most Cadence DSP coprocessors, through the use of alignment registers, provide some support for unaligned accesses. However, using the alignment registers is less efficient than handling aligned references. Often, references are aligned, but the compiler cannot prove they are aligned. You can help improve the result through a combination of command-line options and `#pragma` directives in the source code.

Some of the compiler issues with alignment are highlighted using a vector sum example:

```
int a[N], b[N], c[N];

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

In this example, `a`, `b` and `c` are global arrays and the compiler easily determines that the loop is vectorizable. The XCC and XT-CLANG compilers align all arrays (global and local) to a boundary compatible with the vectorized data types. The vectorizer takes into account the alignment of the array and the lower bound of the loop (in this case, 0) and generates regular, aligned vector loads. Using HiFi 3 and XCC compiler as an example, the inner loop of the generated code looks like this in the assembly file:

```
ae_l32x2.ip      aed0,a2,8; ae_add32 aed1,aed1,aed0
ae_s32x2.ip      aed1,a4,8
ae_l32x2.ip      aed1,a3,8
```

Now consider a version of the vector sum function that takes the arrays as formal parameters:

```
void sum(int *a, int *b, int *c, int n)
{
    int i;
    for (i=0; i<n; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Assuming that you relax the memory model by using a proper aliasing option (see [Aliasing](#) on page 55), the vectorizer will be able to transform the loop. However, as it does not have enough information about the alignment of the arrays, it will assume that they are not aligned. If you are using XCC compiler, you can see the compiler transformation of your code as discussed earlier. An extract from the `.w2c.c` file generated by using the `-clist` command on this example follows. While the inner loop is as efficient as the provably aligned example, there is a significant outer loop overhead in using alignment registers.

```
ali_adr_2 = (_UINT32) (&(b)[0]);
A_ = ae_int32x2_aligning_load_prime(ali_adr_2);
ali_adr_5 = (_UINT32) (&(c)[0]);
A_0 = ae_int32x2_aligning_load_prime(ali_adr_5);
ali_adr_8 = (_UINT32) (&(a)[0]);
A_2 = ae_int32x2_aligning_store_prime();

for(i = 0; i <= (n + -2); i = i + 2)
{
    ae_int32x2_aligning_load_post_update_positive(V_00, A_, ali_adr_2);
    ae_int32x2_aligning_load_post_update_positive(V_, A_0, ali_adr_5);
    V_0 = AE_ADD32(V_00, V_);
    V_1 = V_0;
    ae_int32x2_aligning_store_post_update_positive(V_1, A_2, ali_adr_8);
}
ae_int32x2_aligning_store_flush_positive(A_2, ali_adr_8);
```

For other examples, there may not be a sufficient number of alignment registers, resulting in less efficient code in the inner loop as well as the outer loop. The XT-CLANG compiler yields similar code.

There are several ways to inform the compiler that arrays are correctly aligned and allow it to improve a loop's vectorization. The most local and restrictive way of doing this is through a pragma directive in the source code. If you are sure that a certain pointer refers to a location aligned at a vector boundary, you can specify this in the scope where the pointer is declared using:

```
#pragma aligned (<pointer_id>, <alignment_byte_boundary>)
```

The `alignment_byte_boundary` must be a power of 2, and you cannot use this directive for global pointers.

For example, assuming that `a`, `b`, and `c` are aligned at 8-byte boundaries, this example can be modified as follows to allow better vectorization:

```
void sum(int *a, int *b, int *c, int n)
{
    #pragma aligned (a, 8)
    #pragma aligned (b, 8)
    #pragma aligned (c, 8)
    int i;
    for (i=0; i<n; i++) {
        a[i] = b[i] + c[i];
    }
}
```

Note that `#pragma aligned` can only be used in a function block and the pointer should be defined in this block. Cadence recommends the demonstrated usage.

If you are using XCC compiler, alignment assumptions can also be changed through these two command-line options:

```
-LNO:aligned_pointers=on (the default is off)
-LNO:aligned_formal_pointers=on (the default is off)
```

The first option instructs the compiler to treat *all pointers* used as array bases as if they are aligned correctly for use with the vectorizer. The exact alignment required depends on the target processor's configuration. If you use this option, it is your responsibility to ensure that the alignment constraints are met, otherwise the compiled program may behave incorrectly.

The second option tells the compiler to treat *all pointers passed as formal parameters to functions* as if they are aligned correctly for use in the vectorizer.

Note that the compiler automatically aligns all global arrays correctly. In addition, for users of the default Cadence runtimes, XTOS and XOS, and for most other operating systems supported, local arrays and arrays returned from malloc are also aligned correctly. Therefore, it is usually safe to use the pointer alignment options as long as the program does not contain any pointer arithmetic. Users using other runtime libraries or operating systems not supported by Cadence must ensure that their stack is sufficiently and that their version of malloc sufficiently aligns all arrays. To force alignment during declaration of arrays you can use the following attributes after the name of the declared variable.

XCC:

```
__attribute__ ((aligned (n)))
```

XT-CLANG:

```
__attribute__ ((aligned (n)))
```


or

```
__attribute__((align_value(n)))
```

3.1.8.3 Controlling Optimization Through Pragmas

Various pragmas can be used to control optimization of loops. Typically the pragmas are inserted before the loop under study to hint to the compiler more information that may help in various heuristics to improve vectorization and scheduling. Both XCC and XT-CLANG compilers support similar techniques but with different syntax as outlined below.

3.1.8.3.1 Common Compiler Pragmas

This section described pragmas common to both XCC and XT-CLANG compilers.

#pragma concurrent

When the compiler is unable to resolve the data dependences in an otherwise vectorizable loop, `#pragma concurrent` allows the designer to mark the loop to indicate that each iteration of the loop is independent of all other iterations. This pragma will often make a loop vectorizable. `#pragma concurrent` is placed just before the loop's `for` statement, as shown below:

```
void copy (int *a, int *b, int n)
{
    int i;
    #pragma concurrent
    for (i = 0; i < n; i++)
        a[i] = b[i];
}
```

You must be careful to only use `#pragma concurrent` in cases where the loop iterations are independent. Otherwise, the use of this pragma might change the behavior of your program.

#pragma simd_if_convert

Either compiler is able to vectorize loops containing conditionals using a technique called if-conversion. Using this technique, on some processors all the operations inside a conditional are executed unconditionally, but the results are committed conditionally using conditional move instructions. Consider the following example compiled with ConnX B20 DSP and autovectorization enabled:

```
#pragma simd_if_convert
for (i = 0; i < len; ++i)
    r[i] = c[i] > cond ? a[i] : b[i];
```

Without the pragma the compiler cannot vectorize and complains about “Unsafe accesses in if-statement.” Inserting the pragma changes this and the compiler can now vectorize the code using if-conversion, as shown in the following pseudo assembly code where the operations are now N-multiple-way vector:

```
bbe_la2nx8_ip v0, u0, a2
{ bbe_la2nx8_ip v0, u1, a3; nop; nop; bbe_ltnx16 vb0, v3, v0 }
bbe_la2nx8_ip v1, u2, a4
{ bbe_sa2nx8_ip v2, u3, a5; nop; nop; bbe_mov2nx8t v2, v1, v0, vb0 }
```

In every iteration of the loop data is loaded, the condition is checked and the contents are stored to the output based on satisfying the condition or not.

Note that if the condition is never true, addresses are referenced that would not have otherwise been loaded or stored. If the condition is guarding against a `NULL` pointer dereference, for example, the process of if-conversion might cause a memory exception. Therefore, the compiler will not by default vectorize a conditional unless it can prove that all memory addresses accessed under the conditional are always accessed regardless of the value of the conditional move instruction.

In many cases, the programmer knows that it is perfectly safe to speculatively load or store from these memory addresses, but the compiler cannot be sure. For such cases, the use of `#pragma simd if_convert` by the designer instructs the compiler to perform the if-conversion optimization even if the optimization may generate memory references to otherwise unreferenced addresses. For XCC compiler, the use of the pragma is similar to the use of the `-TENV:X=4` option, except that it applies only to the immediately following loop.

Some processors, such as the ConnX family of cores, have support for predicated load and store instructions, for aligned loads and stores. These processors do not need to unconditionally load or store data in order to vectorize a loop. Therefore, for aligned loads and stores, the use of the pragma or the compiler switch should not make a difference. The pragma and compiler switch will still make a difference when the loop contains potentially unaligned loads or stores as in the example above.

#pragma simd

Use of `#pragma simd` is equivalent to the use of both `#pragma concurrent` and `#pragma simd if_convert`.

3.1.8.3.2 Additional XT-CLANG Pragmas

Additional optimization techniques are available with the XT-CLANG compiler. The following table shows some examples of pragmas that can be inserted before the loop of interest to guide the compiler to achieve better optimization. Note that the target processor/platform will determine the effectiveness of these directives.

Table 6: Pragmas for Better Optimization

Optimization	XCCPragma
Force vectorization on following loop depending on target processor	<code>#pragma clang loop vectorize(enable)</code>
Set the vectorization width based on the target processor resources	<code>#pragma clang loop vectorize_width(n)</code>

3.1.8.4 Speculation (XCC Compiler only)

When using the XCC compiler, performance can sometimes be improved by allowing the compiler to speculate operations, generating code that will be invoked even if the original semantics specify that the code is not invoked. Consider the following example.

```
int main()
{
    int i;
    for (i=0; i<100; i++) {
        if (cond[i] > 0) {
            a[i] = b[i] + c*d;
        }
    }
}
```

The code, as written, will multiply `c*d` 100 times. It would be much more efficient to compute `c*d` once, outside of the loop bounds. If the variables are all integral, there is no harm in speculatively doing the multiply since the multiply has no side effects. If however, the variables are floating point, the multiply operation might set an exception flag. If the application is checking the flags, speculation might not be desired. If the application is not, speculating is advantageous.

The compiler flag, `-TENV:X=n` controls whether the compiler is allowed to speculate operations.

`n=0`: No speculation is allowed.

`n=1`: Only speculation of operations without side effects is allowed.

`n=2`: Allow speculation of floating point compute operations. Do not speculate either integer or floating point divides.

`n=3`: Allow speculation of integer and floating point divides.

`n=4`: Allow speculation of loads. If a load is to an inaccessible address, use of this flag might cause a load exception.

At optimization level `-O2` or below, XCC will default to `n=1`. At `-O3`, XCC will default to `n=2`.

3.1.8.5 Features and Limitations of the Vectorizer

Both XCC and XT-CLANG compilers can produce vectorized code when targeting a SIMD-capable DSP processor platform. The degree of auto-vectorization support varies between the two compilers because they use different heuristics and have different maturity level. However, good code practices to enhance chance of vectorization for either compiler case are for the most part common and described below.

Related Links

[Aliasing](#) on page 55

3.1.8.5.1 For XCC

To use XCC's vectorization feature effectively, it is helpful to know and understand some of its limitations. This feature works best when the program is written in a vectorizable form. Some programs may need to be rewritten to use a different algorithm or different data organization to take full advantage of the vectorizer. The most common limitations relate to aliasing as described in the earlier [Aliasing](#) on page 55 section. Other limitations and features follow.

Limitation in the Stride of Memory Accesses

The vectorizer is only capable of vectorizing loops where successive memory accesses are to nearby locations. An array access of the form `[i+1]`, where `i` is the loop variable, is called a stride-1 access because successive accesses to the array are one element apart. Stride-1 accesses can be mostly easily vectorized since the memory system can load an entire vector of successive elements using a single load. The vectorizer is also capable of vectorizing loops with small positive strides by issuing multiple loads and then using select instructions to extract the appropriate data. Large strided references cannot currently be vectorized.

Choosing which Loop to Vectorize

The vectorizer is capable of vectorizing outer loops as well as inner loops. When there is more than one choice, the vectorizer will use a cost model to decide which loop to vectorize. Often only one loop contains stride-1 or small strided references, and the vectorizer can only choose that one.

Guard Bits

The XCC vectorizing feature does not obey the standard C/C++ integer overflow semantics when targeting several coprocessors including the ConnX, Fusion, HiFi, and Vision family of DSPs. For example, the ConnX BBE16EP register files contain additional guard bits to allow for increased precision on intermediate arithmetic: 40 bits for `int` data types. Results are saturated to 32 bits when stored back to memory. The vectorizer automatically uses the extra precision, thereby changing the behavior of programs that would otherwise suffer from overflow.

3.1.8.5.2 For XT-CLANG

Similar limitations apply to the XT-CLANG case for memory accesses and guard bits as those tend to be dependent on the vector instruction capability of the target coprocessor in use rather the compiler itself.

In terms of vectorization of nested loops, XT-CLANG follows different cost analysis to determine loop vectorization, vector predication and other heuristics so there may be differences in the degree of vectorization for complicated loops. A series of pragmas were discussed earlier to control behavior of these directives.

3.1.8.6 Vectorization Analysis Report

To better explore quality of code and degree of vectorization, the Xtensa Xplorer includes a vectorization assistant tool that can be used to display feedback messages from the compiler during vectorization attempt phase for each of your loops. Same messages are also available as text printouts on the console as well as annotation mark links in the C perspective in Xplorer.

To enable autovectorization you need optimization level -O3 and you must use the `-LNO:simd` command-line option to make the compiler print a summary report for the vectorization analysis to the standard output. The summary report shows the vectorized loops, and indicates the issues that prevent vectorization for non-vectorized loops. This option is available in both compilers for legacy compatibility. In XT-CLANG case, `-fvectorize` compile option is an additional way to invoke vectorization.

Xplorer automatically adds `-LNO:simd_v` to the compile line of any file if vectorization is selected and presents the information in the *Vectorization Assistant* view. Below we show some examples for each compiler.

3.1.8.6.1 XCC Compiler

Analysis information is reported using the following format:

```
<source_file_name>:<source_line_number> (<simd_message_id>):  
<explanation>
```

Consider the following example compiled for the ConnX BBE16EP coprocessor using `xt-xcc -O3 -LNO:simd -LNO:simd_v -c:`

```
void alias(int *a, int *b, unsigned char *c, int n)  
{  
    int i;  
    for (i=0; i<n; i++) {  
        a[i] += b[i];  
    }  
    for (i=0; i<n; i++) {  
        c[i]++;  
    }  
}
```

The following analysis file is generated by the compiler:

```
t.c:2: note:(SIMD_PROC_BEGIN): Vectorization analysis for function 'alias'.
t.c:5: note:(SIMD_ARRAY_ALIAS): Array base 'a' is aliased with array base 'b' at line 5.

t.c:4-5: note:(SIMD_LOOP_BEGIN): Vectorization analysis begins with a new loop.
t.c:5: note:(SIMD_NO_VECTOR_TYPE_SIZE): The processor configuration does not support 16-way
vector int.
t.c:4: note:(SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by 8 (next smaller
vector length)
t.c:5: note:(SIMD_ARRAY_DEPENDENCE): Dependence between array access 'b' and array access 'a'
at line 5.
t.c:4: note:(SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by 4 (next smaller
vector length)
t.c:5: note:(SIMD_NO_REDUCTION): The processor configuration does not support vector reduction
of unsigned int load.
t.c:4: note:(SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by 2 (next smaller
vector length)
t.c:5: note:(SIMD_NO_REDUCTION): The processor configuration does not support vector reduction
of unsigned int load.
t.c:4: note:(SIMD_LOOP_NON_VECTORIZABLE): Loop is not vectorizable.

t.c:7-8: note:(SIMD_LOOP_BEGIN): Vectorization analysis begins with a new loop.
t.c:8: note:(SIMD_NO_VECTOR_TYPE): The processor configuration does not support vector
unsigned char.
t.c:7: note:(SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by 8 (next smaller
vector length)
t.c:8: note:(SIMD_NO_VECTOR_TYPE): The processor configuration does not support vector
unsigned char.
t.c:7: note:(SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by 4 (next smaller
vector length)
t.c:8: note:(SIMD_NO_VECTOR_TYPE): The processor configuration does not support vector
unsigned char.
t.c:7: note:(SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by 2 (next smaller
vector length)
t.c:8: note:(SIMD_NO_VECTOR_TYPE): The processor configuration does not support vector
unsigned char.
t.c:7: note:(SIMD_LOOP_NON_VECTORIZABLE): Loop is not vectorizable.
```

The first loop in the example is not vectorizable because arrays a and b are potentially aliased. The second loop is not vectorizable because the ConnX BBE16EP coprocessor does not support vectorization of operations of type `unsigned char`.

The alias messages are emitted by the data dependence analysis phase of the compiler. Since this phase is applied to the whole function before any optimizations, you may see many alias messages for a function.

Each message shows the line numbers and the names of the two aliasing array accesses. One of the accesses in the message must be a store operation. Occasionally, the compiler does not have the original name of a variable and prints an anonymous name instead.

After the dependence analysis, the vectorizer is invoked on each loop. For each loop, the analysis messages begin with `SIMD_LOOP_BEGIN` and ends with `SIMD_LOOP_VECTORIZED` or `SIMD_LOOP_NOT_VECTORIZED`.

To save compilation time, the vectorizer stops analyzing a loop as soon as it finds a problem. Hence, you will only see one problem per loop.

[Vectorization Messages](#) on page 72 lists the analysis messages currently produced by the vectorizer.

3.1.8.6.2 XT-CLANG Compiler

As with XCC, the analysis information is reported using a similar format:

```
<source_file_name>:<source_line_number> (<simd_message_id>):<explanation>
```

Consider a similar example as in XCC case compiled for the ConnX B20 coprocessor using `xt-clang -O3 -LNO:simd`, or alternatively you can use `-fvectorize` instead to request SIMD optimization.

```
void alias(int *a, int *b, unsigned char *c, int n)
{
    int i;
    for (i=0; i<n; i++) {
        a[i] += b[i];
    }
    for (i=0; i<n; i++) {
        c[i]=3*c[i];
    }
}
```

The following analysis file is generated by the XT-CLANG compiler:

```
remark: testB20.c:1002:0: (SIMD_PROC_BEGIN): Vectorization analysis for function alias.
remark: testB20.c:1004:2: (SIMD_LOOP_BEGIN): Vectorization analysis begins with a new loop.
remark: testB20.c:1004:2: (SIMD_LOOP_VECTORIZATION_TRY): Trying loop vectorization by 16
remark: testB20.c:1004:2: (SIMD_LOOP_VECTORIZED): Loop is vectorized by 16.
remark: testB20.c:1007:2: (SIMD_LOOP_BEGIN): Vectorization analysis begins with a new loop.
remark: testB20.c:1007:2: (SIMD_LOOP_VECTORIZATION_TRY): Trying loop vectorization by 64
remark: testB20.c:1007:2: (SIMD_LOOP_VECTORIZATION_RETRY): Retrying loop vectorization by
32(next priority vector length)
remark: testB20.c:1007:2: (SIMD_LOOP_VECTORIZED): Loop is vectorized by 32.
```

The first loop in the example is vectorizable by 16 as can be seen. The second loop fails to vectorize for the initial size attempted. A second try with a smaller SIMD size finds suitable vectorization length. Sometimes you can force vectorization by using the pragmas we discussed earlier to see if the compiler heuristics were too restrictive and there is benefit in vectorizing a particular loop.

For each procedure the analysis starts with `SIMD_PROC_BEGIN`, and the vectorizer is invoked on each loop. For each loop, the analysis messages begin with `SIMD_LOOP_BEGIN` and ends with `SIMD_LOOP_VECTORIZED` or a reason for failure, for example `SIMD_LOOP_COST_MODEL`.

3.1.8.7 Vectorization Messages

The vectorizer produces a number of analysis messages. Although similar, the XCC and XT-CLANG compilers have different naming conventions for the explanation of vectorization results. Following is an example of an analysis message for the XCC compiler.

SIMD_ACCESS_GAPS Gaps in memory access sequence (base %s1). The compiler is unable to vectorize array accesses with gaps. A loop can only be vectorized if it can be transformed so that it accesses sequential array elements on each loop iteration. The example loop below cannot be vectorized because it accesses only the even elements of the arrays.

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

For an exhaustive list of messages, for XCC refer to the *Xtensa® C and C++ Compiler User's Guide*, and for XT-CLANG refer to the *Xtensa® XT-CLANG Compiler User's Guide*.

3.1.9 Software Pipelining

For many applications, particularly DSP code, performance can be dominated by the performance of inner loops. The compiler will try to software pipeline every inner loop using unrolling techniques. The user is typically able to control the software pipelining and unrolling factor with pragmas as explained below.

XCC Compiler

To software pipeline a loop, the compiler needs to come up with an ordering, or schedule, for all the operations in the loop. Finding the optimal ordering is an NP-complete problem, meaning that it is impossible to always find the best schedule in a tractable amount of time. Instead, the compiler uses heuristics to try many but not all potentially good orderings. Looking at an inner loop, you may feel that the compiler should have scheduled it better. Sometimes you are wrong, sometimes the problem is unrelated to scheduling, but sometimes the problem is simply that the compiler did not find the best schedule.

With super software pipelining, you can ask the compiler to exhaustively, but intelligently, search all possible schedules. To use this feature, add the following `#pragma super_swp` immediately preceding the inner loop. In many cases, the compiler will complete quickly, but in some cases the compiler will never complete. The process can be made somewhat faster by guiding the compiler and telling it how much to unroll the inner loop and how many cycles

(after unrolling) to try to schedule. The following code is an example where the software pipeliner is asked to not unroll an inner loop and schedule it in 28 cycles.

```
#pragma super_swp ii=28, unroll=1
for (i=0; i<n; i++) {
    ...
}
```

Even with the additional hints, there are examples where the compiler will attempt to compile essentially forever.

If the compiler succeeds in a large, but tractable, amount of time, you might not want to repeat the search every time you recompile your application. Instead, compile the code with the additional `-SWP:Op_Info=1` option. With this option, the compiler will place inside the generated `.s` file a string such as the one below following the unroll factor value:

```
#pragma swp_schedule ii=3, unroll=1, sched[4]= 0 1 2 5
```

Moving this pragma into the source program immediately preceding the loop will allow the software pipeliner to again find the schedule very quickly. If the schedule is no longer valid, perhaps the code has been changed, the software pipeliner will try its normal heuristics, but it will use the unroll factor specified in the pragma and will only try schedules at least as long as the one specified in the pragma..

XT-CLANG Compiler

XT-CLANG includes a different implementation of the software-pipelining scheduler compared to XCC, and may result in performance differences.

Programmers have control with pragmas to specify unrolling for software pipelining. Use the following pragmas for setting unrolling preferences, such as enable, fully unroll a loop, do not unroll, or set an unroll factor:

```
#pragma clang loop unroll(enable)
#pragma clang loop unroll(full)
#pragma nounroll
#pragma no_unroll
#pragma clang loop unroll_count(n)
```

Of course breaking the loop manually to keep it manageable or unrolling by hand may still give the best performance in most cases. But these pragmas can help decide if a certain optimization direction is beneficial or not.

3.2 General Coding Guidelines

Previous sections discuss utilizing the compiler to more effectively improve the performance of your application. This section contains a series of general coding guidelines.

3.2.1 Avoid Short Scalar Datatypes

Most Xtensa instructions operate on 32-bit data. There are no 16-bit addition instructions. The system emulates 16-bit addition using 32-bit arithmetic instructions and this sometimes forces the system to convert a 32-bit value into a 16-bit one by extending the sign of a 16-bit result to the upper bits of a register. Consider this example:

```
int foo(short a, short b)
{
    short c;
    c = a+b;
    return c;
}
```

You might expect that the routine would use a single add instruction to add a and b and place the result in the return register. However, C semantics say that if the result of a+b overflows 16 bits, the compiler must set c to the sign-extended value of the result's lower 16 bits, leading to this code: ⁴

```
entry a1,32
add.n a2, a2, a3
slli a2, a2, 16
srai a2, a2, 16
retw.n
```

Much more efficient code would be generated if c was declared to be an `int`. In general, avoid using `short` and `char` for temporaries and loop variables. Try to limit their use to memory-resident data. An exception to this rule is multiplication. If you have 16-bit multipliers and no 32-bit multipliers, make sure that the multiplicands' data types are 8-bit or 16-bit types.

⁴ The code below assumes the configuration does not have the `sext` instruction. If the configuration had the `sext` instruction, the two shift instructions would be replaced with a single `sext` instruction.

3.2.2 Use Locals Instead of Globals

Global variables carry their values throughout the life of a program. The compiler must assume that the value of a global might be used by calls or by pointer dereferences. Consider this example:

```
int g;
void foo()
{
    int i;
    for (i=0; i<100; i++){
        fred(i,g);
    }
}
```

Ideally, `g` would be loaded once outside of the loop, and its value would be passed in a register into the function `fred`. However, the compiler does not know that `fred` does not modify the value of `g`. If `fred` does not modify `g`, you should rewrite the code using a local variable as in [Figure 9: Replacing Locals With Globals](#) on page 75. Doing so saves a load of `g` into a register on every loop iteration.

```
int g;
void foo()
{
    int i, local_g=g;
    for (i=0; i<100; i++){
        fred(i,local_g);
    }
}
```

Figure 9: Replacing Locals With Globals

Alternatively, if the function `fred` does not read or write any global variables other than its function arguments, you can mark the function with the `pure` attribute as show in [Figure 10: Pure Attribute](#) on page 75. If the function `fred` reads but does not write global variables, you can instead using the `const` attribute. For this example, both `const` and `pure` will eliminate the load. In other examples where the variable is written in the calling function, the use of `pure` will eliminate a store but `const` will not.

```
int g;
void __attribute__((pure)) fred(int, int)
void foo()
{
    int i;
    for (i=0; i<100; i++){
        fred(i,g);
    }
}
```

```
}  
}
```

Figure 10: Pure Attribute

3.2.3 Use Arrays Instead of Pointers

Consider a piece of code that accesses an array through a pointer such as in this example:

```
for (i=0; i<100; i++)  
    *p++ = ...
```

In every iteration of the loop, `*p` is being assigned, but so is the pointer `p`. Depending on circumstances, the assignment to the pointer can hinder optimization. In some cases it is possible that the assignment to `*p` changes the value of the pointer itself, forcing the compiler to generate code to reload and increment the pointer during each iteration. In other cases, the compiler cannot prove that the pointer is not used outside the loop, and the compiler must therefore generate code after the loop to update the pointer with its incremented value.

Especially in XT-CLANG case, pointer increment (`ptr++`) of local restrict pointers may upset some optimization. Hence, it is good practice is to avoid such use of local restrict pointers. In general it is safer to use arrays rather than pointers as shown below.

```
for (i=0; i<100; i++)  
    p[i] = ...
```

3.2.4 Minimizing Conditionals

As mentioned in [Using Profiling Feedback](#) on page 50, taken branches impose cycle penalties depending on the pipeline. The Xtensa NX pipeline incorporates branch prediction that somewhat mitigates the problem versus LXtensa X pipeline; however, proceed with caution to avoid branches in code critical sections. The problem is even worse for branches inside of loops because the compiler is typically unable to schedule operations across branches. For performance-critical regions of code, there are several techniques that can be used to mitigate the effects of branches.

Avoid Conditionals

Keep your code simple and avoid conditionals that are not needed. Consider the example in [Figure 11: Conditional Multiplication](#) on page 77. A conditional is used to avoid doing a multiplication in the case that a multiplicand is zero. Unless your configuration has no hardware multiplier, checking for zero will be more expensive than actually doing the

multiplication. The simple code, shown in [Figure 12: Unconditional Multiplication](#) on page 77, will perform better.

```
for (i=0; i<n; i++)
    if (a[i] != 0)
        c[i] += a[i] * b[i];
```

Figure 11: Conditional Multiplication

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

Figure 12: Unconditional Multiplication

Consider the example in [Figure 13: Conditionals that Can Be Replaced by Lookups](#) on page 77. The variable `result` is set to different values depending on the value of `a`. Instead of a series of conditionals, you can create a 16-element lookup array to hold the value of `result` for each possible value of `a` as shown in [Figure 14: Lookups Instead of Conditionals](#) on page 77. The use of the lookup array is a trade-off, requiring more memory, but less time.

```
if (a == 1) result = 2;
else if (a == 7) result = 5;
else if (a == 15) result = 20;
else result = 0;
```

Figure 13: Conditionals that Can Be Replaced by Lookups

```
const char look[16] = {    0, 2, 0, 0,
                          0, 0, 0, 5,
                          0, 0, 0, 0,
                          0, 0, 0, 20};
...
result = look[a];
```

Figure 14: Lookups Instead of Conditionals

Order Conditionals

Given a series of conditional statements such as in [Figure 13: Conditionals that Can Be Replaced by Lookups](#) on page 77, the compiler will order them in the order given. If the variable `a` is usually 15, for example, check for the value 15 first. That way the checks against the other values will not be executed.

Switch Statements

Given a `switch` statement, the compiler has a complicated set of rules for best optimizing the switch. If the number of cases is small, the compiler will implement the switch using a series

of `if` statements to check for every value in series. If the number is large and dense (most values within a range are used), the compiler will use a jump table. Otherwise, the compiler will do a binary search using a series of `if` statements. The actual code generated will differ when compiling for space versus compiling for speed. Therefore, for a series of conditionals based on a single value, it is usually better to use a `switch` statement than to write the code manually using `if` statements. However, when using a `switch` statement, it helps to tell the compiler which conditions are more common. This can be done by compiling with feedback optimization as described in [Using Profiling Feedback](#) on page 50. If this is not possible, for a `switch` with a small number of cases, it might be better to order the cases so that the more frequent ones occur first.

There are some tradeoffs of switch statements over if-cascades. As mentioned, if the values are dense and/or numerous, switch should perform better. If a user however wants to insert complicated frequency hints, a series of if statements is necessary. Otherwise, if the code will be smaller with a switch, it is usually better to use a switch.

Use `#pragma frequency_hint`

Every taken branch incurs cycle penalty, but branches that are not taken do not incur any penalty. If the compiler knows whether a conditional is usually true or usually false, the compiler can try to rearrange the code to minimize the number of taken branches. As described in [Using Profiling Feedback](#) on page 50, you can use pragmas or profiling feedback techniques to provide the compiler with information about branches.

Related Links

[Using Profiling Feedback](#) on page 50

3.2.5 Passing Function Parameters

Consider a situation where you want to write a function that computes the value of some variable, `x`, in the caller. You can either have the function return a value and assign the result of the function to `x`, or alternatively, you can pass the address of `x` into the function and have the function assign `*x` directly inside the function. It is better to have the function return a value. By passing the address of a variable into the function, the compiler must assume that the address is saved away by the function and any pointer dereference anywhere in the program might actually change the value of `x`.

Similarly, scalar variables should always be passed by value. Passing the address of a scalar variable forces the compiler to conservatively assume that the address of the variable is saved by the function.

In contrast, if a structure or class is large enough, it should be passed by reference rather than by value. A structure passed by value must be completely copied on entry to a function. How large is large enough? Unfortunately, it depends. You must trade-off the overhead from the copying versus the limitations caused by poorer compiler analysis when passing pointers. Typically structures of three or fewer fields should always be passed by value in performance-critical code.

Avoid variable arguments. While potentially convenient, variable argument facilities are not efficient.

3.3 Floating Point

Cadence offers multiple options for implementing floating point calculations.

Floating Point Implementation in LX Pipeline

In base processor configurations, floating point computations are emulated using the integer functional units. Typical emulation times for base operations can be from significantly lower than integer multiplier operations.

Cadence offers a double precision floating point acceleration option. Performance for the base operations is improved for a modest number of gates. Divide performance is improved significantly. This option will also improve the emulation time of integer division when compiling code with the `-mcoproc` option.

Cadence also offers a scalar hardware single-precision or double-precision IEEE 754 standard compliant floating point coprocessor. With this coprocessor, most base single (or double) precision floating point operations are fully pipelined, meaning that you can issue one every cycle. Note that the C and C++ language require that a floating point computation be promoted to double precision if any of the operands or results are of type `double`, and if you only have the single-precision coprocessor, double precision arithmetic is emulated using integer operations whether or not you have the single-precision coprocessor. This is frequently an issue with the use of literals, which are by default double precision. Adding the value `1.0` to a single precision variable gets translated by the compiler into a 64-bit emulation routine rather than a single, single-precision floating point add instruction. To avoid this problem either add the suffix `F` to all single precision literals or compile with the flag `-fsingle-precision-constant`.

Note that with the traditional ABI, floating point function arguments are never passed in floating point registers. If you pass a floating point variable to a function, the compiler will move the variable into the integral AR registers before the call and move them back inside the function. With the optional Hardware Floating Point ABI, arguments are passed directly in floating point registers.

Cadence offers several DSP coprocessors, such as ConnX, Vision, Fusion, and HiFi DSP families, with optional floating point units. These are typically vector floating point units (VFPU) compatible with but not identical to the core scalar hardware floating point options. For example, each one uses a DSP coprocessor register file rather than a dedicated register file for floating point variables. All such coprocessors only support the Hardware Floating Point ABI. The VFPUs follow the IEEE 754 standard of floating point conventions.

Both XCC and XT-CLANG compilers, when compiling at optimization level -O3, will by default transform floating point code in ways that might not be bit-exact with the original code. Consider the following example:

```
for (i=0; i<100; i++)
    p[i] = p[i] / n;
```

Rather than issue a divide every iteration of the loop, the compiler might compute the reciprocal of *n* outside of the loop and then just multiply that reciprocal by *p[i]* inside of the loop. The resultant code will be much faster but less accurate. If you need bit-exact code, compile with *-fno-unsafe-math-optimizations*. Similarly, if you are using -O2 but don't care about bit-exactness, you can compile with *-funsafe-math-optimizations*.

Floating Point Implementation in NX Pipeline

As of the current release, the NX pipeline does not offer a scalar core floating point coprocessor to be used with a controller. Users who are interested in hardware floating point acceleration could consider the vector DSP offerings from the Vision and ConnX families of DSPs (for example Vision Q7, ConnX B10, ConnX B20). Each of these DSPs allow configuration options that enable IEEE compliant vector floating point units (VFPUs) with half and single precision support. Future product releases will make available various other configuration options to cover scalar and double precision floating point acceleration offerings.

3.4 C++ Language

C++ is gaining greater acceptance as a language suitable for embedded systems. When used effectively, C++ offers many software engineering advantages with no performance penalties. When used ineffectively, C++ can cause severe bloat, performance penalties and can even result in unreadable and unmaintainable code. It is well beyond the scope of this guide to teach effective programming in C++. However, this section will briefly describe the performance implications of some basic C++ features that might cause you problems.

3.4.1 Streams

C++ encourages using streams for I/O rather than the traditional `printf` from the C library, although the traditional `printf` is also supported. The stream library is very large. A traditional hello program using `printf` requires about 35KB using the `newlib` library and 20KB using `xclib`. In contrast, the stream version shown below requires 286KB with the `newlib` library and 56KB using `xclib`.

```
#include <iostream>
using namespace std;
int main(int ac, char **av)
```



```

{
    cout << "Hello world" << endl;
    return 0;
}

```

If your application is large, the overhead of a library might not matter. However, for small applications, the use of the stream library can dominate the code size of the application.

3.4.2 Exception Handling

Choosing an approach handling error exceptions

C++ exceptions allow a program to throw an exception which is caught by a try-block higher in the call chain, thus allowing a jump across function boundaries. C++ exceptions can replace the traditional technique of checking every function call for errors and returning the error further up to one's own caller. The use of exception handling entails overhead even if exceptions are not used. Consider the example shown below. In the leftmost box, the C++ function `fred` calls the function `wilma`, which may or may not contain code that might throw an exception. In the right most box, is a C-like variant of the code. The call to `wilma` checks for the return of an error value and if it finds one, passes the error value (-1) back to its caller and also cleans up by deleting the locally allocated variable `f`.

<pre> class foo; void fred() { foo myFoo; foo *f = new foo; try { foo->wilma(); } catch (exception &e) { delete f; throw fredException; } ... } </pre>	<pre> class foo; int fred() { foo myFoo; foo *f = new foo; if ((err=foo->wilma()) < 0) { delete f; return err; } ... } </pre>
---	--

When looking at the call to `wilma`, the C++ compiler may not know whether or not the call contains code will throw an exception. Modern C++ allows the function to be marked as `noexcept` or `noexcept(true)` (preferred, C++11/14/17), and `throw()` (C++11/14/17).

If the call to `wilma()` does throw an exception, the compiler must clean up the local variable `myFoo`, by calling its destructor during the processing of the exception. The compiler does this by creating special tables in the binary containing code to call the appropriate destructors relevant for each call site. The optimizer must conservatively hinder optimization to safely handle a potential throw. Since the variable `f` is explicitly allocated, its is not automatically destroyed nor deallocated. The exception must be handled via the `try/catch` statement to not leak memory.

In the case that an exception is actually thrown, the C++ code is somewhat slower than the C variant, but that should not matter much since thrown exceptions should not be common. In the case that an exception is not actually thrown, the C++ code might even be faster than the C code since no comparison is required. However, the C++ code pays some overhead for every function call while the C code only pays the overhead for code that is explicitly checking the return value. Similarly, the C++ code will be larger, particularly in the common case that many C applications do not check every function call.

The XCC compiler does not enable exception handling by default, while the XT-CLANG compiler does include exception handling if not disabled. If you use exception handling, you will by default get an error message from the compiler, and you must then explicitly enable exception handling with `-fexceptions`.

NOTE: The C++ runtime libraries are built with exceptions enabled. This will add a certain amount of extra code to your application. For information on building your own custom version of the a runtime library, contact Cadence support.

3.4.3 Run-Time Type Identification (RTTI)

Choosing whether to identify aggregate types at runtime

With the addition of some overhead to each type and some objects, C++ can identify object type at runtime. In this way, varying ways of handling different objects can be determined without explicit ID's.

XCC does not enable RTTI by default, while XT-CLANG does. Since RTTI does add overhead, some customers elect not to use RTTI. However, some overhead will still be present, since the C++ runtime libraries are compiled with RTTI enabled.

NOTE: for information on building a custom version of the C++ runtime libraries, contact Cadence customer support

3.4.4 Templates

C++ templates allow the writing of generic functions, functions that are parameterized by type. A much simplified example of a templated traditional linked list data structure is shown below. The template defines a list of arbitrary type `T`. Normally, generic functions would be defined to manipulate the list; for example, adding or deleting an element. The variable declaration for `my_int_list`, instantiates a list of integers. If the list template contains a function, `Append`, and that function is invoked on the variable, `my_int_list`, the compiler will create a version of the function `Append` for the `int` datatype. Templates allow you to write one piece of code that is used for multiple datatypes.

Since instantiation happens at compile time rather than at run time, template functions can be more efficient than manually written code that attempts to handle multiple types. However, because the preprocessor creates a version of an instantiated function for every type that is instantiated, the indiscriminate use of templates often leads to greatly increased code size. On the other hand, templates can allow more efficient storage, plus better optimization and

inlining based on type. Consider overloading key template methods and functions to take advantage of type awareness.

```
template <class T>
class LIST
{
    T data;
    class LIST *next;
};

LIST<int> my_int_list;
```

3.4.5 Virtual Functions

Virtual functions can be used together with C++ inheritance. Given a base class and a set of other classes derived off of the base class, a different implementation of a virtual function can be defined for every derived class. Given a variable declared but not defined as the base type, when a virtual function is applied to that variable, the compiler will invoke the version of the virtual function for the actual derived type. The compiler implements virtual functions by creating a table of function pointers. At run time, the actual function address for the derived type is obtained by indexing into the table. With virtual functions, every function call is an indirect call and suffers the overhead of indirect calls.

3.5 Benchmark Example

In this section, some of the techniques that we discussed will be applied to a more realistic example widely used in industry: the EEMBC CoreMark benchmark that measures performance of microcontrollers. You can download the workspace to build and execute the code in Xplorer from XPG server under samples folder for the specific release of tools in which you are interested. We will try the benchmark with incrementally more capable cores adding or subtracting hardware acceleration capabilities.

We started investigation of the CoreMark benchmark performance with the sample_controller configuration from LX pipeline builds found on XPG server view in Xplorer under example-cores folder in the Tutorial sample cores subfolder. We use XCC compiler for the compilation and profiling. We first removed the instructions options for hardware 16 and 32 bit multiplies and 32 bit division support to use as a base for comparison and rebuilt the configuration to compare with the default configuration that includes those options. Obviously the stripped down base configuration without acceleration hardware is smaller but also less capable. We first compiled the application using the -O2 optimization level. A large majority of the time is spent in emulating multiplications since hardware support was removed for the base configuration.

Function Name	Function (%)	Function (F)	Children (C)	Total (F+C)	Called	Size (bytes)
core_bench_list	16.75	3,908,177	19,140,688	23,048,865	60	923
cmp_complex	0.23	54,536	18,954,902	19,009,438	3,342	25
calc_func	1.19	279,516	18,673,714	18,953,230	6,684	232
core_bench_matrix	0.11	27,410	14,467,770	14,495,180	120	113
matrix_test	16.73	3,904,200	10,562,580	14,466,780	120	1,262
__mulsi3	45.35	10,579,719	0	10,579,719	282,588	124
core_bench_state	14.39	3,356,760	818,804	4,175,564	120	1,342
crcu32	3.50	816,824	1,320	818,144	1,920	181

Figure 15: Base CoreMark Profile

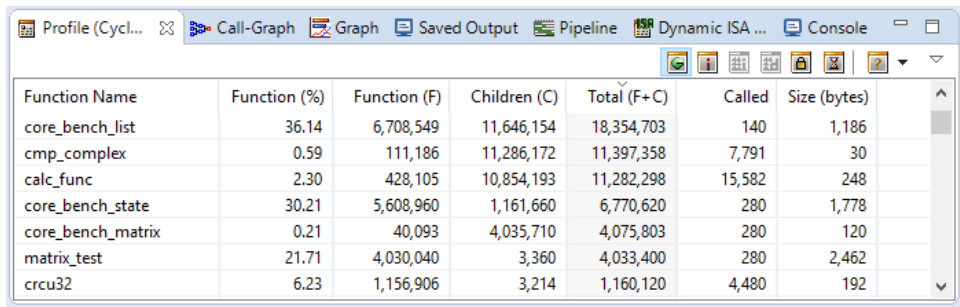
We then revert back to the sample_controller configuration that includes 16/32-bit multiplier hardware and reran the application. The results can be seen in the image below:

Function Name	Function (%)	Function (F)	Children (C)	Total (F+C)	Called	Size (bytes)
core_bench_list	34.16	6,513,858	12,324,367	18,838,225	100	923
cmp_complex	0.47	90,868	12,014,396	12,105,264	5,569	25
calc_func	2.44	465,562	11,546,062	12,011,624	11,138	232
core_bench_state	29.34	5,594,600	1,364,576	6,959,176	200	1,342
core_bench_matrix	0.23	45,686	4,536,250	4,581,936	200	113
matrix_test	23.77	4,532,400	2,200	4,534,600	200	1,320
crcu32	7.13	1,361,276	2,200	1,363,476	3,200	181

Figure 16: Adding a Multiplier

Performance has significantly improved in most benchmark functions. Note that the hardware acceleration for multiplication shows up in many computational intensive routines such as the matrix test that shows more than 200% improvement in cycles with minimal gate additions.

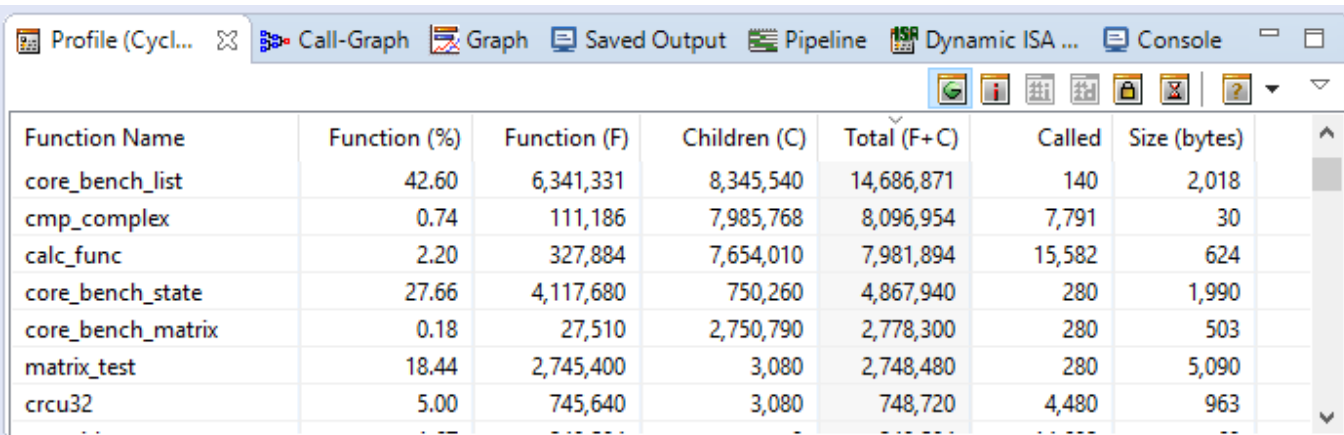
Next we try to see if VLIW or FLIX techniques can help in this example. The sample_flix configuration is then used found in the same XPG subdirectory; this configuration also includes the hardware for multiplication but in addition it allows for parallel issuing of instructions by adding the FLIX feature. Consequently, if there are opportunities to parallelize operation issue in each cycle by bundling them into long instructions, then this feature should provide further benefits. The results are shown below. Note an approximately 10% improvement in cycles for the profiled benchmark functions. However, there is a tradeoff: the code size of most functions has increased since now the compiler uses longer FLIX instructions to bundle the parallel operations. As always, the programmer should weigh the benefits of cycle performance versus code size increase based on the application of interest.



Function Name	Function (%)	Function (F)	Children (C)	Total (F+C)	Called	Size (bytes)
core_bench_list	36.14	6,708,549	11,646,154	18,354,703	140	1,186
cmp_complex	0.59	111,186	11,286,172	11,397,358	7,791	30
calc_func	2.30	428,105	10,854,193	11,282,298	15,582	248
core_bench_state	30.21	5,608,960	1,161,660	6,770,620	280	1,778
core_bench_matrix	0.21	40,093	4,035,710	4,075,803	280	120
matrix_test	21.71	4,030,040	3,360	4,033,400	280	2,462
crcu32	6.23	1,156,906	3,214	1,160,120	4,480	192

Figure 17: Compiling Using FLIX Configuration

The next optimization to try is to elevate to -O3 compilation level. In most cases, but not always, this should provide further improvement from the -O2 level. In addition, this is the only level that allows for autovectorization (SIMD) capabilities if the compiler manages to group scalar elements into vectors, and the configuration used has SIMD vector operations in hardware. For our case, we use the same sample_flix config that does not support SIMD processing but we would like to see if the -O3 level further improves on the -O2. The following figure shows our results.



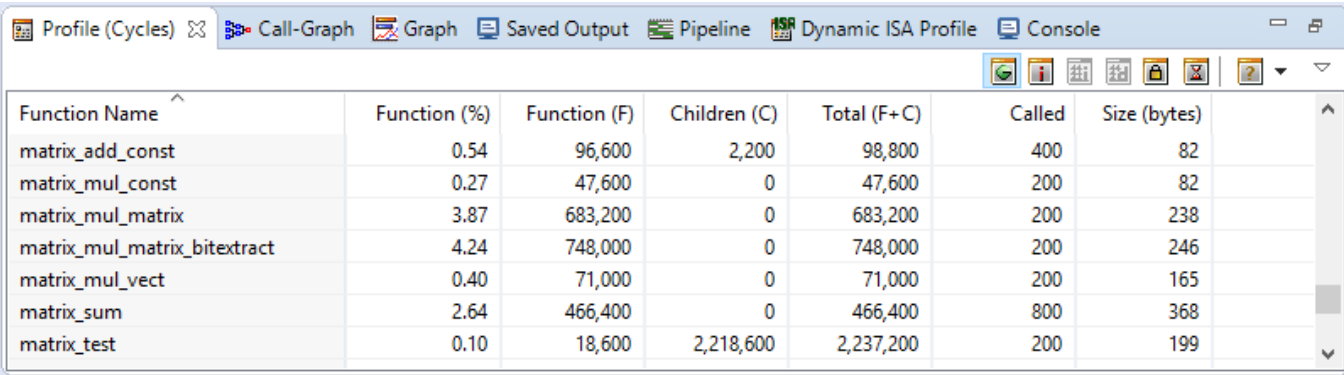
Function Name	Function (%)	Function (F)	Children (C)	Total (F+C)	Called	Size (bytes)
core_bench_list	42.60	6,341,331	8,345,540	14,686,871	140	2,018
cmp_complex	0.74	111,186	7,985,768	8,096,954	7,791	30
calc_func	2.20	327,884	7,654,010	7,981,894	15,582	624
core_bench_state	27.66	4,117,680	750,260	4,867,940	280	1,990
core_bench_matrix	0.18	27,510	2,750,790	2,778,300	280	503
matrix_test	18.44	2,745,400	3,080	2,748,480	280	5,090
crcu32	5.00	745,640	3,080	748,720	4,480	963

Figure 18: Optimizing with -O3 Level

A further approximately 30% improvement is seen with this higher optimization level. Typically the programmer needs to request -O3 if vectorization is desired and supported by the configuration, and in that case the additional directive -LNO:simd also must be included during compilation. Otherwise, experimentation can be tried as to which of the -O2 and -O3 level gives better performance. The performance would depend on many factors and would usually give variation per loop in the schedule produced so it is good to try both.

Similar experimentation can be done with the use of XT-CLANG compiler. We try the last code profile at -O3 level and sample_flix config with FLIX and hardware multiply features as

before but this time we switch to the XT-CLANG compiler. The following figure is the results for the matrix related functions for comparison with XCC.



The screenshot shows the 'Profile (Cycles)' window of the XT-CLANG compiler. It displays a table with the following columns: Function Name, Function (%), Function (F), Children (C), Total (F+C), Called, and Size (bytes). The table lists several matrix-related functions and their performance metrics.

Function Name	Function (%)	Function (F)	Children (C)	Total (F+C)	Called	Size (bytes)
matrix_add_const	0.54	96,600	2,200	98,800	400	82
matrix_mul_const	0.27	47,600	0	47,600	200	82
matrix_mul_matrix	3.87	683,200	0	683,200	200	238
matrix_mul_matrix_bitextract	4.24	748,000	0	748,000	200	246
matrix_mul_vect	0.40	71,000	0	71,000	200	165
matrix_sum	2.64	466,400	0	466,400	800	368
matrix_test	0.10	18,600	2,218,600	2,237,200	200	199

Figure 19: Optimizing with XT-CLANG Compiler

Note that the XT-CLANG compiler uses different heuristics and methods for deciding on inline functions, so some functions are no longer visible independently for cycle counts compared with the XCC list. In order to disable inlining you can use `-fno-inline-functions` compile option. For example, the profile will now show individual functions that can be compared with the XCC function results (if in XCC case they were not inlined either). A common function available for both is the `matrix_test` function, where we see another 20% improvement using the XT-CLANG compiler. The XT-CLANG compiler does not have the longer development history and maturity of the XCC compiler, so some features may still not be available; but significant improvements are expected in the coming releases and eventually it should become the compiler of choice longer term to follow recent industry trends.

4. Processor Configuration Options

The power of the configurable and extensible Xtensa processors is the ability to tailor your processor features (or choose a pre-tailored processor) to match your application. Using TIE language allows the greatest freedom in customizing your processor by extending the existing Instruction Set Architecture (ISA). However, even without TIE, you can configure your processor for your application or application domain.

Depending on whether you will use the Xtensa LX or Xtensa NX pipeline, refer to the *Xtensa LX Microprocessor Data Book* or *Xtensa NX Microprocessor Data Book* for instruction set configuration and memory system options that are directly related to application development.

Note that most of these options change the generated hardware. However, it is easy to perform quick, software-only experiments using the instruction set simulator before settling on a final configuration.

Processor configuration options are described in the *Xtensa LX Microprocessor Data Book* and *Xtensa NX Microprocessor Data Book*.

5. Tensilica Instruction Extension (TIE) Language Programming

The Tensilica Instruction Extension (TIE) language describes extensions to the core instruction set documented in the Xtensa Instruction Set Architecture (ISA) Reference Manual.

The Tensilica Instruction Extension (TIE) language gives you great power to customize your processor for an application or an application domain while still maintaining a high level programming model.

Refer to the *Tensilica® Instruction Extension (TIE) Language User's Guide* and the *Tensilica® Instruction Extension (TIE) Language Reference Manual* for details on this topic.

6. Ensuring Software Quality

Topics:

- *Using C library console and file I/O support for testing*
- *Code Coverage*

The Xtensa tools provide ways to examine your code for issues and to measure aspects of your software. These include:

- Console and file I/O during test execution
- Code coverage - whether and how often functions and lines of code have been executed during tests

6.1 Using C library console and file I/O support for testing

The C/C++ runtime libraries included with Xtensa tools support console and file I/O. This allows your test code to accept input and send output that can be verified automatically. You can also pass standard command line arguments when you launch your test program.

The simplest approach is to use console I/O. The usual stdin/stdout/stderr streams supported but may need BSP support to work on hardware targets. Explicit file I/O is also available. This allows larger data sets to be input and output for processing, and for reports to be printed directly to a file. Limited support is available on hardware targets, as described below.

Software simulation with ISS and XTSC

The instruction set simulator (ISS) is executed standalone (`xt-run`) or via the debugger (`xt-gdb`). With `xt-run`, command line arguments are provided after the program path. With `xt-gdb`, command line arguments for the program to be run can be provided via the “run” command or the “set args” command. In both cases, the console for `xt-run`/`xt-gdb` is used for console I/O. Both `xt-run` and `xt-gdb` can be run from scripts to create complex and automated test cases.

This gets more complicated if you are simulating via XTSC. You may have multiple processors being simulated. Also, console I/O will include output from the SystemC simulation itself so verifying test results might be difficult with standard I/O. Consider using explicit file I/O to avoid confusion.

In Xplorer, the integrated debugger includes a console window in the Xplorer interface when switched to the Debug perspective. The console shows both stdout and stderr output and can accept input to be relayed to the running program. The interface is the same for ISS execution or for remote execution via remote debug. For more information about debugging with Xplorer, please see the Xplorer built-in help.

I/O Support on the Simulator (ISS)

When running on the ISS, I/O operations are handled by the simulator. File I/O is targeted to the host's file system (the machine on which the ISS is running). Console I/O goes through the `xt-run` console. The low level I/O functions (`open`/`close`/`read`/`write` etc.) trap to the simulator which then performs the needed actions.

Simulation is paused during I/O processing, so the time overhead of the simulator's handling of I/O operations does not affect program timing. However, the part of the processing handled by the C library does consume processing cycles.

Simulator I/O support is linked in automatically if needed when using the “sim” LSP or related LSPs targeted to the simulator.

Remote I/O Support with xt-gdb

When debugging a remote (hardware) target, xt-gdb provides file I/O support using the GDB Remote Serial Protocol. Both console and file I/O are supported. File I/O is targeted to the host's file system (the machine on which xt-gdb is running). Console I/O goes through the xt-gdb console. The low level I/O functions (open/close/read/write etc.) trap to the debugger which then performs the needed actions.

The communication between xt-gdb and the target is synchronous, and the target is halted while xt-gdb performs operations on its behalf. This can significantly change program timing when I/O is performed.

The protocol only supports regular files. Special devices, pipes, sockets etc. are not supported.

Debugger I/O support requires the libgdbio library to be linked in. This is automatic if the "gdbio" LSP is used. The library may be explicitly linked by specifying "-lc -lgdbio" on the link command line. This library if used on the ISS can automatically detect the presence of the ISS and switch to using simulator traps for I/O.

BSP-Supported I/O Operations on hardware

When executing on hardware targets, I/O may be performed to local devices (e.g. a UART) if supported by the Board Support Package (BSP). Typically, a minimal BSP might provide a single UART to act as the console. More sophisticated BSPs might provide access to local or remote filesystems. Keep in mind that such I/O is likely to be synchronous and could significantly affect program timing.

Providing BSP support requires implementing several low-level functions needed by the C library. Please refer to the Xtensa C library (xclib) and Newlib documentation for details.

Differences between ISS and Remote Targets

It is important to note some differences between file I/O behavior on the ISS and the behavior on remote targets supported by xt-gdb. The difference in program timing due to I/O has already been mentioned.

Another significant difference is in the behavior of the `fopen()` function. If you use `fopen()` to open a file, it is translated to the system call `open()`. If you link your program with libgdbio and it runs the `open()` function on a hardware target, xt-gdb will open the file on behalf of the target program. It always opens files in binary mode even if the host OS running xt-gdb is not Linux. For example, `fopen(..., "w")` will be translated implicitly to `fopen(..., "wb")` when running a gdbio-based executable in xt-gdb connected to a hardware target.

However, when running the same code on the ISS, the text/binary mode is not forced and the mode supplied by the `fopen()` call is passed along to the underlying host OS. When `fopen()` is called with just "w", the file is opened in text mode, which means that it will behave differently depending on the host OS that is running the ISS. On Linux, there is no difference between opening a file in text mode and opening it in binary mode, but on

Windows, there can be differences, such as the addition of a LF character (0x0a) after every CR character (0x0d).

This can be confusing if the same executable using the libgdbio library is run on the ISS and a hardware target, and the file ends up being opened differently. To be on the safe side it is preferable to open files in binary mode (e.g. "wb").

6.2 Code Coverage

Code coverage analyzes execution of your program to provide information about what parts of the code have been exercised.

Xplorer has a Feedback mode which instruments the code so that statistics on execution can be collected. Many test programs can be run and the line-for-line execution statistics are aggregated. You can then open the Code Coverage view from Xplorer to see the result.

Note that there are two different procedures used for XCC and XT-CLANG.

7. Memory Layout

Topics:

- [Using Local Memories Only](#)
- [Using Local Memories for Some Code and Data](#)

Cadence core architectures provide great flexibility in designing your memory subsystem. You can create caches of many different types and sizes. You can also create local instruction or data memories that can be accessed by the processor in a single cycle. Given a memory subsystem design, the programmer needs a way to place different pieces of code and data at different addresses.

The system software developer may have complicated needs with regards to memory layout. Handlers must be placed at specific addresses; the memory space of devices must be reserved against general usage. These uses are beyond the scope of this guide and are instead covered in the *Xtensa Linker Support Packages (LSPs) Reference Manual*.

Sometimes you want to dynamically load an application into memory. If you are using an operating system, Linux, for example, the operating system may have sophisticated mechanisms for dynamically loading programs and sharing libraries across programs. For users running without an operating system, Cadence has tools to help, but they are also beyond the scope of this guide. These tools are covered in detail in the *Xtensa System Software Reference Manual*.

This guide focuses on the needs of application developer; how they place code or data in local IRAM or DRAM memory.

7.1 Using Local Memories Only

By default, your application will not use local memories. If your configuration contains no caches, every load, store, and instruction fetch will go to system memory and take many cycles. In the simplest case, you may want to use only local memories. There are several ways to achieve this goal.

First, if you are developing using the default, `sim`, or the `min-rt` LSP (linker support package), you can instead use the `sim-local` or `min-rt-local` LSP. For command-line users, simply link your application with the flag `-mlsp=sim-local` or `-mlsp=min-rt-local`. For Xplorer users, simply select the appropriate LSP from the Linker tab of the Xtensa Project Build Properties dialog box.

If you are using a different LSP, the easiest way on the command line is to create an entire new set of lsp's that default to use local memories with the tool `xt-regenlsp` as follows:

```
xt-regenlsp -dstbase dir -mlocalmems
```

Once created, the new LSPs can be used on the command line using the flag `-mlsp=dir/lspname` or in Xplorer by selecting a custom LSP.

Alternatively, from Xplorer, right-click the configuration and you can select an option to create, or clone, a new configuration that defaults to using all local memories.

7.2 Using Local Memories for Some Code and Data

Most users do not want to put everything in local memories, only important code and data. For code that you are writing, a function or variable can be put into local memories using attributes. For example, to put a function `foo` into the single, local IRAM, declare it as follows:

```
extern void foo(void) __attribute__((section(".iram.text")));
```

Make sure that the declaration appears in the same file as the function definition and precedes the definition. For configurations with two local IRAMs, the predefined sections `.iram0.text` and `.iram1.text` are associated with each one respectively. The section `.iram.text` is an alias for `.iram0.text`.

Similarly, global variables can be placed into the local DRAM as follows:

```
int globvar __attribute__((section(".dram.data"))) = 0;
```


For configurations with two local DRAMs, the predefined sections `.dram0.data` and `.dram1.data` are associated with each one respectively. The section `.dram.data` is an alias for `.dram0.data`.

The compiler may create read-only data to support the compilation of a particular function. For example, the compiler may create a read-only table to implement a C switch statement. If a function is sufficiently important to place in IRAM, the generated jump table might be important enough to go in DRAM. The following attribute on a function declaration tells the compiler to both allocate the function in IRAM and any generated read only data in DRAM.

```
extern void foo(void) __attribute__((section(".iram.text"), rodata_section(".dram.rodata")));
```

Note that variables are not affected by the `rodata_section` attribute, only compiler generated data. Variables must be explicitly attributed.

Local variables and dynamically allocated variables cannot be allocated to memories using attributes; they always reside in the `stack` and `heap` respectively. The stack and heap can be moved to a local DRAM using the Memory Map editor in Xplorer or on the command line using the `xt-regenlsp` tool. See the *Xtensa Linker Support Packages (LSPs) Reference Manual* for details.

Sometimes you want to place code that you did not write into the IRAM or DRAM. A common example is all or part of the C library. This can be done using *xt-objcopy*.

For example, to place any code pulled from the C library into instruction RAM, place any associated literals (if using PC-relative literals) into data RAM, and leave other sections in their default location. You can do this by renaming sections within the C library as follows:

```
mkdir tmp
xt-objcopy --rename-section .text=.iram0.text \
  --rename-section .literal=.dram0.literal \
  <xtensa_root>/xtensa-elf/lib/libc.a tmp/libc.a
```

where `<xtensa_root>` is the location of your Xtensa core package. Telling the linker to use an alternate version of a library or object file specified by the LSP `specs` file, such as the C library, is best done using the `-L <searchdir>` flag to XCC or XT-CLANG compiler. This tells the linker to look in the specified directory ahead of its standard search path. Place your alternate library in a separate directory free of other unwanted files, and add that directory to the linker search path, for example, `-L tmp` for the above code.

The link order tool described in [Memory System](#) on page 28 allows an automated way to map the most important functions, based on profile information, into the IRAM.

8. Devices and Synchronization

Topics:

- *Memory Ordering*
- *Cache Coherence*
- *Volatile Devices or Memory*

The C/C++ programming model mostly assumes that everything your program does is expressed directly in C/C++ or indirectly via C libraries. In reality, many programs need to communicate with external devices or with other processors. When communicating with the outside world, the programmer needs control to ensure that memory references communicating with the outside world actually happen, that memory references happen in the intended order and that multiple processors are able to atomically read or write multiple blocks of memory. Such control can be difficult to achieve when compilers reorder memory references or delete unused variables, when processors cache data without any attempt to keep multiple caches coherent and when multiple processors can attempt to write to the same locations in arbitrary orders. While the C/C++ language gives a little help in the form of *volatile* attributes, that help is coarse and incomplete. Instead, the programmer must go outside the bounds of the language when extra control is needed.

8.1 Memory Ordering

C/C++ provides a sequential programming model in which every statement happens in the order written. In reality, to improve performance, the compiler can change the order, and in particular, the compiler can change the relative order of two memory references that refer to distinct locations. Similarly, because the Xtensa processor is pipelined and contains internal buffers, the hardware might also change the relative order of two memory references, as seen by an outside agent, whenever the two memory references refer to different memory locations. Normally, these optimizations are safe, as well as effective. However, in a real system with multiple processor cores or independent devices, at times you may want to preserve memory ordering in certain portions of your program. For example, in a multiprocessor system, one processor might compute data into shared memory and then set a shared memory flag variable to indicate to another processor that the data is available. If either the compiler or the hardware reorders memory references, the second processor might see the flag being set before the data is actually available.

This section assumes that the memory locations being used for communications are uncached. Caches add an additional level of complexity that is discussed in [Cache Coherence](#) on page 104.

Programmers often try to guarantee sequential semantics by using the `volatile` attribute. This is different from using `volatile` to guarantee that a memory reference to a memory mapped device with side effects is not deleted. Unfortunately, the use of `volatile` for this purpose is inefficient and potentially dangerous. The C and C++ standards guarantee that two volatile references are not reordered with respect to each other, but they do not guarantee that a volatile reference is not reordered with respect to a non-volatile one. Thus, to be safe, both flags and data must be marked as `volatile`, but doing so can be inefficient. Because `volatile` can be used for memory consistency, the compiler is forced to be overly conservative when a strict ordering is not required but `volatile` is used for devices with side effects. The Xtensa hardware may reorder memory references unless separated with a MEMW or other synchronization instruction. Both XCC and XT-CLANG compilers separate all volatile references with a MEMW instruction by default, even though this is usually unnecessary for devices with side effects. The flag `-mno-serialize-volatile` instructs either compiler to omit the MEMW instructions.

A safer and more efficient way to guarantee consistency is through the use of a pragma, as follows:

```
#pragma flush_memory
```

The compiler ensures that all data is effectively flushed to or from memory at the point of the pragma, so that all memory references before the pragma occur before any memory references after the pragma. The compiler also places a MEMW instruction at the point of the pragma to ensure that the hardware does not reorder references across the pragma.

Consider the example below where one processor is signaling to another that data has been produced while the other is spinning waiting for the flag to be set.

<pre>for (i=0; i<n; i++) { data[i] = ... } #pragma flush_memory flag_data_ready = 1;</pre>	<pre>while (!flag_data_ready) { #pragma flush_memory } for (i=0; i<n; i++) { ... = data[i]; }</pre>
---	--

Figure 20: Communicating with Flags

All the data is guaranteed to be written by the first processor before the flag, `flag_data_ready`, is written. None of the data consumed by the second processor is consumed until the second processor sees that the flag has been written.

For examples with a single flag variable used on configurations with the Synchronize Instruction option (Xtensa LX pipeline configuration option), Cadence offers two instructions, `S32RI` and `L32AI`, that allow you to more efficiently combine the effects of storing or loading the flag variable together with the MEMW ordering guarantees in a single instruction. For example, the flag setting example above can be replaced with the code in this example:

<pre>#include <xtensa/tie/xt_sync.h> ... for (i=0; i<n; i++) { data[i] = ... } XT_S32RI(1, &flag_data_ready, 0);</pre>	<pre>#include <xtensa/tie/xt_sync.h> ... while (!XT_L32AI(&flag_data_ready, 0)) { ; } for (i=0; i<n; i++) { ... = data[i]; }</pre>
---	---

Figure 21: S32RI and L32AI

The semantics of the `S32RI` instruction ensure that any prior load or store instructions are externally visible before the value being stored by the `S32RI` becomes visible. Similarly when testing a flag variable, the `L32AI` load instruction can load the flag variable and guarantee that the flag is loaded before any subsequent memory references.

If your external system bus supports AMBA AXI4 and your processor is configured with EA (Exclusive Access) enabled, then you can use `L32EX/S32EX` instructions to synchronize between processors. For more information, see (Refer to *Xtensa® ISA Reference Manual* for details).

8.1.1 TIE Ports

TIE ports (`lookups`, `queue`, `import_wire` and `state` with the `export` attribute) potentially have similar consistency issues. Refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* or the *Tensilica Instruction Extension (TIE) Language User's Guide* for details about these features. You might, for example, use a TIE output queue to produce data and then use a shared memory flag to signal that the data has been computed. By default, the XCC or XT-CLANG compiler assumes that references to different TIE ports are unrelated to each other or to memory, and hence can potentially be reordered. Similarly, by default, the Xtensa hardware might reorder a TIE port reference with respect to another, or to memory in the sense that the effects of a reference from a later instruction might become externally visible before the effects of an earlier reference. Note that neither the compiler nor the hardware will reorder multiple references to the same TIE port. In addition, related TIE ports such as a `QUEUE` and its associated `NOTRDY` interface are considered to be one port, so that references to one of them are not reordered with respect to references to another.

XCC and XT-CLANG compilers provide an option, `-mflush-tieport`, that guarantees that neither the compiler nor the hardware will reorder a TIE port reference with respect to another or to memory. Given this option, the compiler will serialize all TIE port references with respect to each other and to memory, and the compiler will insert an `EXTW` instruction before and after each TIE port reference. Note that `EXTW` may take an arbitrary amount of cycles, as it must wait for all writes to output queues and all TIE lookup accesses to be completed.

The use of the `-mflush-tieport` flag is potentially expensive overkill in terms of slowing down the performance of the application. Instead, Cadence recommends using the `pragma flush`. This `pragma` works similarly to `pragma flush_memory`, except that it also affects the ordering of TIE ports. All memory or TIE port references issued before the `pragma` are guaranteed to complete before any memory or TIE port references issued after the `pragma`. The compiler will insert a single `EXTW` instruction at the point of the `pragma`. As the `pragma` affects memory as well as TIE ports, there is no need to use both `pragmas`.

Note that if you are not using any TIE ports or do not require them to be sequentially consistent, it is more efficient to use `pragma flush_memory`. The use of `pragma flush` generates an `EXTW` instruction rather than the `MEMW` instruction generated with `pragma flush_memory`, and the `EXTW` instruction is potentially more expensive than the `MEMW` instruction.

Note that the use of the compiler option or the `pragma` guarantees that earlier accesses complete before later accesses, from the point of view of the processor. However, there is no way to guarantee that all system effects from earlier accesses are complete. If an output queue is connected to a deep external queue, an `EXTW` instruction will cause the processor to stall until all pushed data enters the external queue, not until all pushed data leaves the other end of the external queue.

TIE Ports and Deadlock

There are potential deadlock conditions when designing or programming a system with queues. If one processor is blocked waiting for a signal from an independent agent, and the independent agent is not sending the signal because it is in turn waiting for some signal from the processor, the system will deadlock. In addition to situations that might inherently lead to deadlock, deadlock might also result because the compiler or hardware reorders references to TIE ports and memory with respect to the original program order.

Consider a simple two processor system as shown in [Figure 22: TIE Ports and Deadlock](#) on page 103. Each processor is sending data to the other via a small queue. The first processor writes into a queue and then tries to get a response back from the second queue. The second processor reads data from the first queue and then sends back a response to the second queue. If the compiler for the first processor rearranges the references so that the first processor tries to read its input queue before it writes its output queue, the read will cause the processor to block while waiting for data before it has a chance to write its output data. The system will deadlock because the second processor will never write its queue until it receives its data from the first processor.

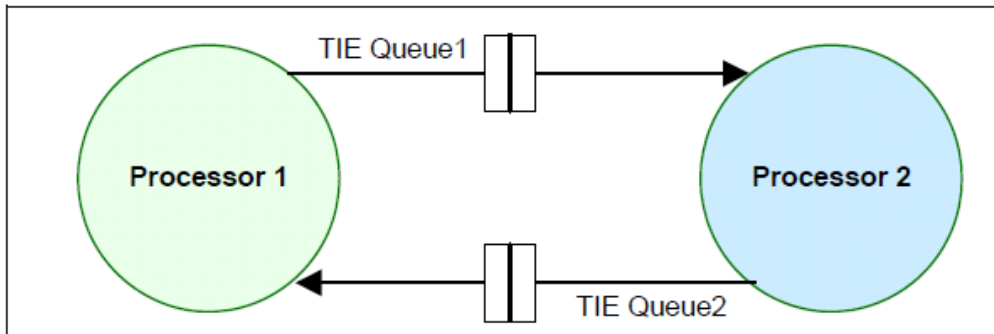


Figure 22: TIE Ports and Deadlock

To avoid this situation, you can add a `#pragma flush` in between the two queue references, or you can use the compiler flag `-mflush-tieport`. Both these choices, however, will cause the compiler to generate an `EXTW` instruction. For this example, you want to prevent the compiler from reordering references so that a later reference will not cause the processor to stall without allowing an earlier reference to complete. However, you do not care if a later reference becomes externally visible before an earlier one. In such situations, an `EXTW` instruction is not necessary. Instead you can use `#pragma no_reorder` or the compiler flag `-mno-reorder-tieport`. These options prevent the compiler from reordering references, but do not generate `EXTW` instructions.

8.2 Cache Coherence

Cadence processors optionally support caches as an easy to use mechanism to improve the memory performance of an application. With caches, memory being accessed is automatically copied into close, fast cache memory, and is only copied back into system memory whenever memory values are changed in locations currently held in the cache.

Devices that are not performance-critical or that are accessed through narrow interfaces are often mapped into uncached memory regions. However, every uncached write generates a bus transaction and every uncached load requires at least six cycles and also generates a bus transaction. Therefore, performance critical devices that access blocks of data are typically either cached or DMA'd into and out of local memories.

Caches on multiprocessor systems without hardware cache coherence complicate the memory consistency issues discussed in [Memory Ordering](#) on page 100. First, caches, at least write-back caches, delay stores an arbitrary amount of time. Second, loads may read a memory value from a local cache even if some other processor or device has already modified the global value. Therefore, on multiprocessor systems, caches may be inconsistent with each other. Consider again the examples in [Figure 20: Communicating with Flags](#) on page 101 and in [Figure 21: S32RI and L32AI](#) on page 101. Let us now assume that the variable *flag* and the array *data* are in cached memory. When the first processor writes the flag, the flag and/or some of the data might stay in the cache for an arbitrary amount of time. Therefore, the second processor might wait forever for the flag to be set or worse, it might read a mixture of old and new data if some but not all of the data written by the first processor has been sent back to main memory. Similarly, when the second processor is spinning, waiting for the flag to be set, if the flag is not set on the first iteration, it will never be set because the cache of the second processor will not get updated. Perhaps worse, if some but not all of the old data remains in the second processor's cache, the second processor will get a mixture of old and new data. To be safe, the first processor must force the data and the flag to be written back to main memory and the second processor must invalidate the flag and the data before reading them.

The Cadence HAL (Hardware Abstraction Layer) software provides functions for managing cached data:

```
// ensure cached data sent back from the cache to system memory
xthal_dcache_line_writeback(void *address);

// ensure cached data at a particular address is deleted from the cache
xthal_dcache_line_invalidate(void *address);
```

. When writing or reading a large amount of data, the functions *xthal_dcache_region_writeback(void *address, unsigned size)* and *xthal_dcache_region_invalidate(void *address, unsigned size)* can be used to writeback or invalidate multiple addresses with a single function call. Further details about the HAL

functions are in the *Xtensa System Software Reference Manual*. The example in [Figure 20: Communicating with Flags](#) on page 101 is rewritten to handle cached flags and data below.

```
#include <xtensa/config/core.h>
...
for (i=0; i<n; i++) {
    data[i] = ...
}
xthal_dcache_region_writeback(data, size);
flag_data_ready = 1;
xthal_dcache_line_writeback(&flag_data_ready);

#include <xtensa/config/core.h>
...
xthal_dcache_line_invalidate(&flag_data_ready);
while (!flag_data_ready) {
    xthal_dcache_line_invalidate(&flag_data_ready);
}
xthal_dcache_region_invalidate(data, size);
for (i=0; i<n; i++) {
    ... = data[i];
}
```

Figure 23: Coherent Use of Flags

An additional problem occurs because data is written to and from cache in units of lines that are anywhere from 16 to 256 bytes. The compiler will place unrelated variables in the same cache line. Imagine a situation where one processor is writing one flag while another processor is writing a different flag, but both flags are next to each other in memory and share a cache line. If each processor sets its respective flag at the same time, the cache of each processor will have its flag set but not the other. Then depending on which cache line gets written back to memory last, only one of the flags will remain set. To avoid such scenarios, you must ensure that a variable being communicated not be shared with any unrelated variables. The easiest way to do that is to both align the variable to the size of the cache line and make sure that the size of the variable is a multiple of the cache line size. Variables can be aligned to 64 bytes, for example, using attributes as follows:

```
int flag __attribute__((aligned (XCHAL_DCACHE_LINESIZE)));
```

8.3 Volatile Devices or Memory

In many cases, you may have a device in uncached memory that is accessed through a load or store to a single, memory mapped location. You may not care if loads or stores to the device are reordered with respect to other memory references. However, you do care that the

device is accessed exactly as many times as specified in the original C/C++ code. You do not want the compiler to move a load from a memory mapped hardware queue outside of a loop, for example, because it appears to the compiler that successive loads must all be loading the same value. For such cases, the use of `volatile` is appropriate.

Note however that the compiler will insert `MEMW` instructions around all `volatile` references because the compiler does not know whether you care about memory ordering. The flag `-mno-serialize-volatile` instructs XCC and XT-CLANG to omit the `MEMW` instructions. When using this flag, the programmer must use explicit serialization if needed.

Note that `volatile` does not force cached data to be flushed to external memory. This requires use of cache operations.

9. Handling interrupts

Topics:

- *Interrupt and Exception environment*
- *Handling exceptions*
- *Handling Interrupts*
- *Communicating with interrupt handlers*

Handling interrupts in C

Interrupts are closely tied to the BSP and to the RTOS or runtime used. Cadence provides two such environments: XTOS and XOS.

9.1 Interrupt and Exception environment

Determining your interrupt environment

The interrupt scheme used by your processor depends on architecture and, for Xtensa LX, a configuration selection. Since interrupts involve both hardware and software, careful decisions made jointly by hardware and system software designers provide the best performance and reliability.

Xtensa LX processors normally are configured with the XEA2 architecture. The number of priority levels and vector locations are defined at configuration time. All Xtensa NX processors use the XEA3 architecture, which is more flexible and allows setting of parameters at runtime.

For XEA2 priority level 1 interrupts, a soft prioritization scheme is provided by XTOS and XOS. Priority levels 2 and above are prioritized by hardware.

With the XEA3 exception architecture used in Xtensa NX processors, the 32-interrupt limit is gone, and each interrupt's characteristics are independently programmed. For portability, both XTOS and XOS API's for interrupts and exceptions have been adapted to control both LX and NX capabilities.

Interrupts are closely tied to the BSP (board support package) and to the RTOS or runtime used. Cadence provides two runtime environments: XTOS and XOS. Your choice will depend on the needs of your application and your desired system architecture. Specific hardware choices also determine how specific interrupt handling is initialized, for example, sensitivity (edge/level). During handling, an edge-triggered interrupt may not require clearing at the source, but a level-sensitive interrupt will repeatedly call the handler until the source is quieted and the input is deasserted.

XTOS provides the environment for a single-threaded program, while XOS allows multiple threads of execution. Each runtime provides its own exception and interrupt management functions. (Refer to *Xtensa® System Software Reference Manual* and *XOS Reference Manual* for details). These functions isolate you from most of the differences between processors and architectures.

To respond to an interrupt or exception, simply register a C function which has the appropriate signature. That function will be called whenever the interrupt or exception occurs. As much as possible, special instructions or acknowledgements required by the processor are provided by the XTOS or XOS framework. You must still deal with the circumstances that caused the event, and pay attention to all details needed for proper handling of the event.

9.2 Handling exceptions

The Xtensa exception mechanism allows special conditions to be handled by separate code immediately when instructions encounter them. Certain exceptions are simply part of normal

operation of the processor, like window overflow/underflow and memory error corrections, and the runtime software usually provides these handlers.

Other exceptions are the responsibility of the application. The default handlers halt the execution of the application or thread. This is useful in development, or when it is simple to reboot the application after it hangs. In most applications, however, a carefully designed response to exceptions is required.

Exception handling is a complex, low-level exercise and requires system programming expertise. See the (Refer to *Xtensa® System Software Reference Manual* for details) for detailed information. Application designers decide on a strategy for recovery, which typically involves directly invoking reset or signaling a host or control CPU to perform recovery.

By default, XOS handles exceptions that occur during thread execution by suspending the thread and saving state. Other threads can access the suspended thread for diagnosis. One approach to recovery is to restart the failed thread. See (Refer to *XOS Reference Manual* for details) for details.

C Exception handlers in XTOS have a specific signature:

```
typedef void (xtos_handler_func)(void *arg);           // definition at the XTOS I/F
```

The argument passed is a pointer to a register save area on the stack. Before a C handler is called for a given exception, working registers and CPU state are saved to this area. When using windowed registers, an extra step is necessary to ensure active registers are spilled to this save area so that they may be accessed:

```
xthal_window_spill();           // spill active registers to stack
```

The saved registers are saved into a structure accessible from the argument passed. You may declare your exception handler like this to access this save area:

```
#include <xtensa/xtruntime-frames.h>           /* Exception stack frame pointer defined here */
void hnd_divzero(UserFrame *frame) {
    regs.pc = frame->pc; regs.ps = frame->ps; regs.exccause = frame->exccause;
}
```

As this code fragment shows, the UserFrame struct provides access to processor state. Use the exccause field to examine details of the particular exception being handled. For more information on the EXCCAUSE register, see (Refer to *Xtensa® ISA Reference Manual* for details).

Since an exception interrupts and possibly restarts an instruction, it is not possible to mask or prevent exceptions occurring during your code's execution. However, the exception handler cannot be interrupted. A second exception during the exception handler results in a double exception, which is an unrecoverable error. Any coordination between the exception and the rest of the code must not use techniques that require critical sections.

9.3 Handling Interrupts

C interrupt handlers in XTOS have a specific signature:

```
typedef void (xtos_handler_func)(void *arg);           // definition at the XTOS I/F
```

The argument passed is a pointer to a register save area on the stack. Before a C handler is called for a given exception, working registers and CPU state are saved to this area. When using windowed registers, an extra step is necessary to ensure active registers are spilled to this save area so that they may be accessed:

```
xthal_window_spill();                                // spill active registers to stack
```

The saved registers are saved into a structure accessible from the argument passed. You may declare your exception handler like this to access this save area

Exception handling is a complex, low-level exercise and requires system programming expertise. See the (Refer to *Xtensa® System Software Reference Manual* for details) for detailed information.

XOS handles exceptions that occur during thread execution by suspending the thread and saving state. Other threads can access the suspended thread for diagnosis. One approach to recovery is to restart the failed thread. See (Refer to *XOS Reference Manual* for details) for details.

Certain exceptions occur under normal situations. For instance, configurations with MMU running Linux use exceptions for demand paging. A debug exception is used for inserting breakpoints. These are normally operating system handlers and do not require any user programming.

9.4 Communicating with interrupt handlers

Without coordination, interrupts can happen while your program is executing sensitive code, such as data shared between handler and the executing program thread, e.g., main function. To prevent these conflicts, you can use a number of techniques

- Disable all interrupts - this might delay other interrupts at a higher priority
- Disable specific interrupt - this might delay the specific handler
- Set interrupt level - setting the interrupt level to N prevents interrupts at that priority and lower
- Use single-writer/single-reader techniques (requires no coordination)
- Use RTOS constructs - message-passing, mutex, semaphore, events, etc.

There are consequences to each of these techniques, beyond the scope of this document. Consult your systems programming team or architect to ensure that your handler will execute properly.

Index

- `__attribute__`
 - `aligned` [105](#)
 - `always_inline` [49](#)
 - `noinline` [49](#)
 - `optimize` [46](#)
 - `pure` [75](#)
 - `section` [96](#)
- `-LNO`
 - `aligned_formal_pointers` [64](#)
 - `aligned_pointers` [64](#)
 - `simd` [17](#), [39](#), [59](#), [61](#)
- `-TENV:`
 - `X=4` [66](#)
- `#pragma aligned` [63](#)
- `#pragma concurrent` [65](#)
- `#pragma flush` [102](#)
- `#pragma flush_memory` [100](#), [102](#)
- `#pragma frequency_hint` [78](#)
- `#pragma no_reorder` [103](#)
- `#pragma simd` [66](#)
- `#pragma simd_if_convert` [65](#)
- `#pragma super_swp` [72](#)

A

- Algorithms and performance [26](#)
- Aliasing and performance [55](#)
- Alignment
 - for vectorization [62](#)
- Analysis
 - messages [72](#)

C

- Cache Explorer [30](#)
- Compiler
 - code quality and performance [36](#)
- Configuration
 - and performance [27](#)
- Configuration Options
 - software selections [20](#)

D

- Deadlock
 - and TIE ports [103](#)
- Diagnosing performance problems [26](#)

F

- Feedback
 - `-fb_create` [50](#)

G

- Global variables and performance [75](#)
- Guard bits [68](#)

I

- Inlining [49](#)
- IPA [46](#)

L

- Linker
 - IPA [47](#)

M

- Memory
 - local memories [96](#)
 - ordering of accesses [100](#)
 - system and performance [28](#)
 - volatile memory consistency [105](#)

P

- Performance
 - algorithms [26](#)
 - aliasing [55](#)
 - arrays and pointers [76](#)
 - compiled code quality [36](#)
 - conditionals [76](#)
 - configuration [27](#)
 - function parameters [78](#)

- Performance (*continued*)
 - global variables [75](#)
 - memory system [28](#), [96](#)
 - short datatypes [74](#)
- Performance microarchitecture [32](#)
- Pipeline delays [32](#)
- pragma
 - aligned [63](#)
 - concurrent [65](#)
 - flush [102](#)
 - flush_memory [100](#), [102](#)
 - frequency_hint [78](#)
 - no_reorder [103](#)
 - simd [66](#)
 - simd_if_convert [65](#)
 - super_swp [72](#)
- Pragmas
 - aligning data [63](#)
 - optimizing loops [57](#)
 - pipelining [72](#)
 - vectorization [65](#)
- Profiling feedback [50](#)

S

- SIMD [59](#)
- Speculation [67](#)

T

- TIE ports [102](#)

V

- Vectorization
 - aligning data [62](#)
 - analysis report [69](#)
 - features and limitations [68](#)
 - messages [72](#)
 - pragmas [65](#)
 - viewing compiled results [60](#)
- Volatile memory consistency [105](#)