Bound-T time and stack analyzer

Application Note

SPARC/ERC32

V7, V8, V8E

Tidorum Ltd
www.tidorum.fi
Tiirasaarentie 32
FI-00200 Helsinki
Finland

This document was originally written at Space Systems Finland Ltd by Sami Saarinen and Niklas Holsti within an ESTEC-supported project to develop a Bound-T version for the ERC32 (SPARC V7) target. The document is currently maintained by Niklas Holsti at Tidorum Ltd.

Document reference:	TR-AN-SPARC-001
Document issue:	Version 7
Document issue date:	2010-02-10
Bound-T/SPARC version:	4a3
Last change included:	BT-CH-0219
Web location:	*http://www.bound-t.com/app_notes/an-sparc.pdf*

Credits:
This document was created with the free OpenOffice.org software, *http://www.openoffice.org/*.

The information in this document is believed to be complete and accurate when the document is issued. However, Tidorum Ltd. reserves the right to make future changes in the technical specifications of the product Bound-T described here. For the most recent version of this document, please refer to the web address *http://www.bound-t.com/*.

If you have comments or questions on this document or the product, they are welcome via electronic mail to the address *info@tidorum.fi*, or via telephone or ordinary mail to the address given below.

Please note that our office is located in the time-zone GMT + 2 hours (+3 hours in the summer) and office hours are 9:00 -16:00 local time.

Cordially,

Tidorum Ltd.

Telephone: +358 (0) 40 563 9186
Fax: +358 (0) 42 563 9186
Web: *http://www.tidorum.fi/*
http://www.bound-t.com/
Mail: *info@tidorum.fi* (please include the word "Bound-T" in the Subject line)

Post: Tiirasaarentie 32
FI-00200 HELSINKI
Finland

Credits

The Bound-T tool was first developed by Space Systems Finland Ltd (*http://www.ssf.fi*) with support from the European Space Agency (ESA/ESTEC). ESA has also supported some of the work on the version for SPARC V8 (specifically the LEON). Free software has played an important role; we are grateful to Ada Core Technology for the Gnat compiler, to William Pugh and his group at the University of Maryland for the *Omega* system, to Michel Berkelaar for the *lp-solve* program, to Mats Weber and EPFL-DI-LGL for Ada component libraries, to Ted Dennison for the *OpenToken* package, and to Marc Criley for the *XML EZ_Out* package. Call-graphs and flow-graphs from Bound-T are displayed with the *dot* tool that is a part of the *GraphViz* package from AT&T Bell Laboratories.

The extension to include SPARC versions V8 and V8E was supported by ESA/ESTEC under ESA/Contract No. 19535/05/NL/JD/jk. The interface to the RapiTime tool was created in collaboration with Rapita Systems Ltd within that contract.

Contents

Index of Tables

Document change log

Issue	*Section*	*Changes*
7	Front matter	Added section for document change log.
7	Section 2.2	Corrected the discussion of caches in the "General " section.
7	Section 5	Extended introduction.
7	Section 5.4	Extended to include return offsets.
7	Section 5.5	Added section on instruction roles (none defined).

1 INTRODUCTION

1.1 Purpose and Scope

Bound-T is a tool for computing bounds on the worst-case execution time of real-time programs; see reference [1]. There are different versions of Bound-T for different target processors. This Application Note supplements the Bound-T Reference Manual [1] and the Bound-T User Guide [2] by giving additional information and advice on using Bound-T for target processors that use the SPARC architecture (version V7 or version V8, including the "embedded extensions V8E).

For timing analysis this version of Bound-T supports one particular SPARC processor that is used in European space projects: the ERC32. Stack usage analysis is supported for all SPARC V7, V8 or V8E processors. This includes the LEON processor family.

The ERC32 is a SPARC V7 processor for space applications implemented by Atmel Wireless & Microcontrollers (formerly TEMIC Semiconductos) (*http://www.atmel-wm.com*). The three-chip implementation of this processor is described in references [4] [5] [6], but it is no longer used; the Bound-T tool supports the single-chip version [7], for which also the former references apply. The ERC32 is usually applied without cache memories.

The LEON is a series of SPARC V8 processors [9] [10] designed by Gaisler Research and implemented for space applications by Atmel as the AT697 chip [11]. LEON processors usually include on-chip cache memories. The cache memories, and other design differences, make the LEON instruction timing different from that of the ERC32.

The first goal of this document is to explain the sort of SPARC code that Bound-T can or cannot analyse and so help you write analysable programs. The second goal is to explain the additional command-line options and other controls that are specific to the SPARC.

Some information in Section 4.3 and Chapter 5 of this Application Note applies only when the target-program executable is generated with specific compilers, linkers and run-time kernels. Other compilers may be addressed in separate Application Notes.

1.2 Overview

The reader is assumed to be familiar with the general principles and usage of Bound-T, as described in the Bound-T Reference Manual [1]. The reference manual also contains a glossary of terms, many of which will be used in this Application Note.

So what's it all about?

In a nutshell, here is how Bound-T bounds the worst-case execution time (WCET) of a subprogram: Starting from the executable, binary form of the program, Bound-T decodes the machine instructions, constructs the control-flow graph, identifies loops, and (partially) interprets the arithmetic operations to find the "loop-counter" variables that control the loops, such as n in "for ($n = 1$; $n < 20$; $n{+}{+}$) { ... }".

By comparing the initial value, step and limit value of the loop-counter variables, Bound-T computes an upper bound on the number of times each loop is repeated. Combining the loop-repetition bounds with the execution times of the subprogram's instructions gives an

upper bound on the worst-case execution time of the whole subprogram. If the subprogram calls other subprograms, Bound-T constructs the call-graph and bounds the worst-case execution time of the called subprograms in the same way.

When does it work?

This sort of "static program analysis" is in theory an unsolvable problem and cannot work for *all* programs (like the well-known "halting problem"). It succeeds for programs that have a suitable structure, for example programs in which all loops have counters with constant initial and final values and a constant step. Moreover, since we are analysing low-level machine coder rather than high-level source code, the nature of the instruction set and the specific instructions used (or, usually, generated by the compiler) may help or hinder the analysis.

This Application Note explains how Bound-T has been adapted to the architecture of the SPARC processors and how to use Bound-T to analyse programs for these processors and in particular for the ERC32. To make full use of this information, the reader should be familiar with the architecture and SPARC architecture and instruction set as presented in references [8] [9] [10].

The remainder of this Application Note is structured as follows:

- Chapter 2 describes the main features of the SPARC architecture and how they relate to the functions of Bound-T.
- Chapter 3 defines in detail the set of SPARC instructions and registers that is supported by Bound-T.
- Chapter 4 explains those Bound-T command arguments and options that are wholly specific to the SPARC or that have a specific interpretation for this processor.
- Chapter 5 addresses the user-defined assertions on target program behaviour and explains the possibilities and limitations in the context of the SPARC and certain software development tools used with the SPARC.

1.3 References

[1] Bound-T Reference Manual.
Tidorum Ltd., Doc. ref. TR-RM-001.
http://www.bound-t.com/manuals/ref-manual.pdf

[2] Bound-T User Guide.
Tidorum Ltd., Doc. ref. TR-UG-001.
http://www.bound-t.com/manuals/user-guide.pdf

[3] SPARC Processor for Space Applications.
TEMIC Semiconductors.

[4] TSC691E Integer Unit User's Manual for Embedded Real time 32-bit Computer (ERC32) for SPACE Applications.
Temic Semiconductors, Rev. I, September 1998.

[5] TSC692E Floating Point Unit, User's Manual for Embedded Real time 32-bit Computer (ERC32) for SPACE Applications.
Matra MHS, Rev. H, December 1996
TEMIC Semiconductors.

[6] TSC693E Memory Controller User's Manual for Embedded Real time 32-bit Computer (ERC32) for SPACE Applications.
Matra MHS, Rev. D, April 1997
Temic Semiconductors.

[7] TSC695E Rad-Hard 32-bit SPARC Embedded Processor User's Manual.
Atmel Wireless & Microcontrollers, Rev. F, March 2001.

[8] SPARC V7.0 Instruction Set, for Embedded Real time 32-bit Computer (ERC32) for SPACE Applications.
Atmel Wireless & Microcontrollers, Rev. C, 28 August 2001.

[9] The SPARC Architecture Manual, Version 8.
Revision SAV080SI9308. SPARC International Inc. 535 Middlefield Road, Suite 210, Menlo Park, CA 94025.

[10] SPARC-V8 Embedded (V8E) Architecture Specification.
Version 1.0, October 23, 1996. SPARC International, 3333 Bowers Avenue, Suite 280, Santa Clara, CA 95054-2913, USA

[11] Rad-Hard 32 bit SPARC V8 Processor AT697E.
Rev. 4226C–AERO–08/05, Atmel Corporation.

[12] ERC32 Timing Measurements.
Space Systems Finland Ltd., BTE-TN-SSF-001, Version 1, September 2001.

[13] Using Bound-T in HRT Mode.
Tidorum Ltd., Doc. ref. TR-UM-002.

[14] RapiTime. *http://www.rapitasystems.com/wcet.html*, Rapita Systems Ltd.

[15] Exporting the Bound-T Program Model to RapiTime. Tidorum Ltd, Doc. Ref. TR-PEAL-ICD-001, Issue 2, 2006-06-02.

[16] DWARF Debugging Information Format, Version 3.
Free Standards Group, December 20, 2005. *http://dwarf.freestandards.org*.

[17] Bound-T Assertion Language.
Tidorum Ltd., Doc. ref. TR-UM-003.
http://www.bound-t.com/manuals/assertion-lang.pdf

1.4 Abbreviations and Acronyms

See also references [1] and [2] for abbreviations specific to Bound-T, reference [8] for the mnemonic operation codes and register names of the SPARC, and reference [13] for abbreviations specific to the "HRT mode" of Bound-T.

ASI	Alternate (address) Space Identifier
ASR	Ancillary State Register
CWP	Current Window Pointer
DWARF	Debugging With Attribute Record Format (although this expansion is not given in the DWARF standard itself [16])
EDAC	Error Detection And Correction
ELF	Executable and Linking Format

ERC32	Embedded Real-time 32-bit Computer
ESF	Execution Skeleton File (for an HRT program)
FP	Floating Point
fp	Frame Pointer = register **r30**
FPQ	Floating-Point Operation Queue
FPU	Floating-Point Unit
GNAT	GNU Ada Translator
HR	Hard Real Time
ILP	Integer Linear Programming
IOP	Internal OPcode
IU	Integer Unit
MEC	Memory Controller
MMU	Memory Management Unit
NaN	Not a Number
ORK	Open Ravenscar Kernel
PSR	Processor State Register
RISC	Reduced Instruction Set Computer
RW	Register Window
sp	Stack Pointer = register **r14**
SPARC	Scalable Processor Architecture
STABS	Symbol TABle
TBR	Trap Base Register
TPO	Threads and Protected Objects file
WCET	Worst-Case Execution Time
WIM	Window Invalid Mask register

1.5 Glossary of Terms

See also reference [1] for general terms used in static execution-time analysis and Bound-T in general. The following list contains only terms specific to the SPARC version of Bound-T as described in the present document.

Alternate	An alternate SPARC calling sequence or return sequence. That is, a sequence that uses the **UNIMP** instruction to transmit an additional parameter.
Caller-save	A subprogram that is called with **CALL** or **JMPL** immediately followed by a **SAVE** instruction in the delay-slot. The calling sequence (**CALL; SAVE** or **JMPL; SAVE**) thus creates a register window for the called subprogram. The subprogram itself shall not contain a **SAVE** but shall contain a **RESTORE**.
Ipoint	An instrumentation point for RapiTime. See reference [14] and section 4.7.

Leaf — A suprogram that executes with the same register-window as its caller. Any normal subprogram (as distinct from a trap handlere) that is neither a *self-save* subprogram nor a *caller-save* subprogram is a leaf subprogram. A leaf subprogram shall not contain any **SAVE** or **RESTORE** instructions and **SAVE** instructions shall not appear (as the delayed instruction) in calls to this subprogram.

Non-leaf — A subprogram that is not a leaf subprogram. See *leaf*. Thus, a subprogram that executes in its own register-window, not in the same register-window as its caller.

Normal — A normal SPARC calling sequence or return sequence. That is, a sequence that does not involve the **UNIMP** instruction.

Self-save — A subprogram that contains one **SAVE** instruction that creates a register window for use in the subprogram. This subprogram shall be called with an instruction pair (starting with **CALL** or **JMPL**) that does not contain a **SAVE** instruction. The subprogram shall contain a **RESTORE** instruction executed after the **SAVE**.

1.6 Typographic Conventions

We use the following fonts and styles to show the role of pieces of the text:

register — The name of a SPARC register embedded in prose.

INSTRUCTION — A SPARC instruction.

-option — A command-line option for Bound-T.

symbol — A mathematical symbol or variable.

`text` — Text quoted from a text / source file or command.

2 THE SPARC AND TIMING ANALYSIS

2.1 The SPARC architecture

The SPARC architecture [8] [9] [10] defines a 32-bit, load-store RISC architecture where all computation is register-to-register. Instruction fetch, decode, execute and write cycles are pipelined, and branch instructions have one delay-slot.

Each instruction is 32 bits wide. Integer operations are typically executed in one cycle. The execution time of some complex instructions depends on the implementation, especially for the optional floating-point instructions which are (conceptually) implemented by an optional floating-point unit running as a coprocessor. Depending again on the implementation the processor may or may not have caches, fetch buffers, write buffers or other performance accelerators. The architecture also defines an optional Memory Management Unit (MMU) for virtual memory addressing and memory protection.

The number of pipeline stages and the pipeline timing may also depend on the implementation. However, two pipeline stages are architecturally visible and are called the “decode” and “execute”. The result is that jumps and calls are usually “delayed”; the jump or call instruction is followed by one “delay” instruction that is executed before the transfer of control.

Data formats and operations

Integer operations work on 32-bit integers, but load/store operates also on data of length 8 or 16 bits (with or without sign extension) or 64 bits. Two's complement representation is used for integer data.

A special variant of integers called *tagged data*, which is useful for dynamically typed programming languages, is also supported by the SPARC. It contains 30 bits of data and a 2-bit tag.

Integer addition and subtraction are supported in hardware by the Integer Unit (IU) [4] but (full) multiplication and division are not. Some implementations may support floating-point operations in hardware by a separate Floating-point Unit (FPU). The ERC32 has an FPU [5] which runs as a concurrent unit with the Integer Unit. The LEON AT697E has an FPU that runs sequentially with the Integer Unit. In some SPARC implementations the FPU can work on only one instruction at a time; other implementations have pipelined FPUs and may have a floating-point instruction queue to further buffer FP instructions streaming from the IU to the FPU.

Floating-point operations use either 32-bit (single precision) or 64-bit (double precision) IEEE-standard formats. Some implementations also support wider formats such as a 96-bit “extended” format (in the ERC32) or the 128-bit “quadruple” precision format.

Memory and register addressing

Memory is byte-addressed but the memory bus is usually 32 bits wide. Data and instructions use the same address space (von Neumann architecture). Instructions must be aligned at 32-bit (4-byte) multiples, and data for load and store must be aligned at a multiple of the data length (*eg.* an even address for 16-bit data).

For integer operations, 32 registers of 32 bits can be accessed, divided into 8 "global" registers (**r0** to **r7**), 8 "out" registers (**r8** to **r15**), 8 "local" registers (**r16** to **r23**) and 8 "in" registers (**r24** to **r31**). The "out", "local" and "in" registers are mapped to a 24-register window into a much larger register file, relative to a *Current Window Pointer* (CWP) in the processor state register (PSR). The normal calling convention manipulates the CWP on call/return so that the "out" registers of the caller are visible as the "in" registers of the callee. The "local" registers are private to each subprogram level. Thus, the register file is used as a ring buffer to hold the local context of the deepest currently active subprograms.

The size of the register file, or register ring, depends on the implementation. The ERC32 and the AT697E have register files with 128 registers (8 full sets or "windows" of 16 registers).

Hardware traps occur when the CWP wraps around in the register ring, and software trap-handlers then spill or load register windows to a memory-resident stack. The number of windows spilled or loaded per trap is not fixed by the architecture, but the manuals [4] state that spilling and loading one window at a time has been found best. One window is reserved for the trap handler. Thus, on the ERC32 or LEON seven windows are usually available for normal subprograms, but a different number can be defined by the *Window Invalid Mask* (WIM) register.

The floating-point unit contains a more conventional set of 32 floating-point registers (**f0** to **f31**) without any register-window mechanism.

The **Y** register is a special 32-bit register in the integer unit. An instruction that multiplies two 32-bit values places the upper 32 bits of the 64-bit product in **Y**. Vice versa, a division instruction can take the upper 32 bits of the 64-bit dividend from **Y**.

Fault tolerance

The ERC32 internal data paths include error checks and the memory includes error correction bits that are updated and checked on the fly by the EDAC (error detection and correction). The LEON design is available in a fault-tolerant version (LEON FT) with similar features.

Optional co-processor

The SPARC architecture defines an interface to an optional co-processor (in addition to the Floating-Point Unit) and a subset of the instruction space for co-processor instructions. The purpose and functionality of the co-processor is left entirely open. The ERC32 and LEON processors do not have such co-processors.

Control flow

The IU PSR contains four condition-code bits, **N** = negative, **Z** = zero, **V** = overflow, and **C** = carry. Many arithmetic and logical instructions have two forms, one that sets condition codes and another that does not. The FPU has its own set of condition code bits. Accordingly, there are two conditional branch instructions: **Bicc** for IU condition codes and **FBfcc** for FPU condition codes. (The SPARC architecture has a third such instruction, **CBccc**, for coprocessor condition codes, but the ERC32 and LEON have no coprocessors.) Each of these has 16 forms that differ by the condition for which the branch occurs, including “always” or “never”.

The instruction in the delay slot (following the branch) is always executed if the branch is taken. An “annul” bit in the instruction selects whether the delay-slot instruction is executed in case the branch is not taken. (The branch-always instruction is a special case here.) Thus, if the annul bit is set, the delay-slot instruction works as if it were the first instruction at the target of the branch.

Concurrent Functional Units

In some SPARC implementations, such as the ERC32, the FPU can execute operations concurrently with the IU, as a form of parallel computation. This requires special consideration in any static execution-time analysis of such SPARC devices. The rest of this chapter first describes the analysis of the IU, and then considers the FPU.

2.2 Static Timing Analysis of the SPARC Integer Unit

General

The SPARC Integer Unit architecture in the ERC32 is very regular. The ERC32 IU is suitable for static analysis by Bound-T, since instruction timing in no case depends on the data being processed, but only on the control flow and sometimes on data-flow dependency between the source and target registers of consecutive instructions. The dependencies can be resolved statically.

Bound-T currently does not include a static analysis of cache memories. Thus, Bound-T cannot analyse execution time for processors with caches, for example the LEON processors. Moreover, the LEON has several other “accelerator” features that the present version of Bound-T does not model, such as the Instruction Burst Fetch, the Write Buffer, and the MMU. The timing model in this version of Bound-T applies only to the ERC32 and does not support timing analysis of LEON processors.

Register Window Traps

The static timing analysis of the Integer Unit is complicated by the spilling and loading of register windows when the register file overflows or underflows. Whether a spill or load occurs for a particular call or return depends on the call-nesting depth, which is a property of paths in the call-tree and cannot be deduced from a local analysis of the subprograms.

To predict the occurrence of register window traps, Bound-T/SPARC optionally applies a global analysis to the call-tree rooted at the subprogram(s) that the user specifies for WCET analysis. The global analysis computes two values for each subprogram *S* in the call tree:

- an upper bound *rwu_max*(*S*) on the number of register windows in use when *S* is called,
- lower and upper bounds on the number *win_depth*(*S*) of register windows pushed and popped by an execution of *S*, including its callees.

These numbers let Bound-T predict which calls and returns can cause overflow or underflow traps. The prediction is conservative (if a trap actually occurs, it is predicted) but not necessarily precise (a predicted trap may not actually occur). The prediction relies on some assumptions:

- Bound-T assumes that only the program itself changes the number of register windows in use, via **SAVE** and **RESTORE** instructions at calls and returns. If the program is run under a real-time kernel that can interrupt or pre-empt tasks, this means that Bound-T assumes that the kernel preserves the contents of the register file so that when the task is resumed, the number of active windows is the same as when the task was interrupted.
- Bound-T assumes that the trap handlers spill or load one register window at a time, as recommended in [4]. These trap handlers are usually part of the kernel or run-time system.

What can be done if the kernel or run-time system violate these assumptions? There seem to be only two reasonable ways for the kernel to handle the register file when a task is resumed: either the register file is entirely restored, as Bound-T assumes, or only one window is restored, which can be expected to minimize task switching overhead. In the latter case, Bound-T's prediction for overflow traps remain safe (but may become more pessimistic), but it predicts too few underflow traps. If W is the number of windows available to the application (as defined by the WIM, usually $W = 7$), each suspension and resumption causes at most W-1 underflow traps in addition to those that Bound-T predicts.

There are two ways to work around this. The first way is to tell Bound-T to assume that *every* return may cause an underflow trap (command-line option *-returns_trap*). This is a safe prediction but quite pessimistic. The second way is to use the standard Bound-T analysis to estimate the WCET of each task, including the predicted number of traps, but to add the time for the additional underflow traps to the kernel's task-resumption time, when analysing the schedulability of the whole (multi-task) program.

If more than one register window is spilled or loaded by the trap handlers, the current Bound-T register-file trap analysis cannot be used. Bound-T then computes WCET values which do not include any time for register file traps. We are considering a potential extension to Bound-T/SPARC to make the number of register windows loaded or spilled a configurable parameter.

When the register-file trap analysis is enabled (option *-rw*) Bound-T by default assumes that the root subprograms, that is, the subprograms named by the user for WCET analysis, start execution with two register windows in use (*rwu_max* = 2). The rationale is that a root subprogram is assumed to be called from another subprogram (perhaps in the kernel), which uses at least one window. Therefore, if the entry sequence of the root subprogram executes a **SAVE** instruction, as is usually the case, it will have at least two windows in use. This assumption can easily be overridden by asserting the value of the property *rwu_max* for the subprogram (option *-assert*).

For each predicted register-file overflow trap, Bound-T includes the WCET of the overflow trap handler in the WCET bound of the *calling* subprogram, even if the **SAVE** instruction that triggers the trap is actually at the start of the *callee* subprogram.

For each predicted underflow trap, Bound-T includes the WCET of the underflow trap handler in the WCET bound of the *returning* subprogram.

By default, Bound-T itself computes the WCET of these trap handlers, locating them with a default trap base address. The user can specify a different trap base address (option *-trap_base*) or directly the WCETs for the trap handlers (options *-rw_overflow* and *-rw_underflow*).

2.3 Static Timing Analysis of the SPARC Floating Point Unit

The static execution-time analysis of the SPARC floating-point unit (FPU) is complicated by two features:

- Possible concurrent execution of the FPU and the IU.
- Possible queuing of several floating-point instructions between the IU and the FPU.
- Variable (data-dependent) floating-point instruction time.

FPU and IU sequential operation

In SPARC implementations such as the LEON AT697E that feature an FPU with sequential operation, all instructions are fetched and executed sequentially. Integer instructions and control instructions are executed in the IU (the FPU is idle); floating-point instructions are executed in the FPU (the IU is idle). The next instruction, of whatever type, starts execution only after the preceding instruction is completed. Of course the IU pipeline overlaps some of the execution steps of consecutive instructions but this overlap is exactly modelled by the delayed control transfers and the "pipeline blocking" delays that occur between certain pairs of instructions.

Thus, a static timing analysis of a SPARC implementation with a sequential FPU can treat IU instructions and FPU instructions as one stream. The only complicating factor is the variable, data-dependent execution time of the floating-point operations, on which more below.

As this version of Bound-T only supports timing analysis for the ERC32 processor, which is always equipped with a concurrent FPU, the sequential alternative is not really relevant.

FPU and IU concurrent operation

In SPARC implementations such as the ERC32 that feature an FPU with concurrent operation, floating-point operations are fetched and decoded by the IU and FPU in lock-step, but the decoded floating-point instructions enter the FPU's Floating-point Operation Queue (FPQ) and are then executed asynchronously, while the IU fetches, decodes and executes further IU instructions. If the IU encounters more floating-point operations, they can be put in the FPQ as long as there is room, while earlier floating-point operations are still executing. (The ERC32 FPQ [5] has room for only one instruction, which simplifies the analysis as shown later.)

The floating-point compare instructions such as **FCMPS** are not executed concurrently with IU instructions, instead they cause the IU to halt until their execution is finished. This ensures that the resulting condition codes are available to the IU for the **FBfcc** instruction.

Load/store instructions for the FPU registers (**f0** to **f31** and control registers) are not placed in the FPQ, but other interlocks apply (see below).

The FPU thus executes concurrently (in parallel) with the IU, with synchronisation only at specific points. The main synchronisation points are the following ([5], section 3.2.2.2):

- When a new floating-point operation is fetched, but the FPQ is full, the IU waits until the currently executing FPU instruction finishes and leaves room in the FPQ for the new one. The waiting time is thus the remaining execution time of the head instruction in the FPQ.
- A "load" into an FPU register is delayed until all queued FPU operations that use this register (as input or output) are completed.

- A "store" from an FPU register is delayed until all queued FPU operations that place results in this register are completed.
- A "load" into or a "store" from the FPU status register is delayed until any ongoing FPU instruction is completed.

A complete model of the FPU state for timing analysis should include the number of operations in the FPQ, their source and destination registers, their worst-case execution time, and the remaining (worst-case) execution time of the instruction in execution (the head of the FPQ). This FPU state should then evolve in the natural way as the execution progresses, with old operations finishing their execution and leaving the FPQ, and new operations entering the FPQ.

For the SPARC devices that Bound-T supports, where at most one operation fits in the FPQ, the model is simplified: either the FPQ is empty, or it contains one operation with known source and destination registers and some remaining execution time (a zero time can be used to indicate that the FPQ is empty). Bound-T analyses this dynamically evolving state to bound the possible states at each point in the control-flow graph of a subprogram.

Variable instruction execution time

The execution time of most floating-point instructions is data-dependent, with sometimes a large factor between the minimum and maximum times. For example, on the ERC32 [5] a single-precision addition (**FADDS**) takes between 4 and 17 cycles, and double-precision square-root (**FSQRTd**) takes from 6 to 80 cycles. This may lead to WCET bounds that are pessimistic.

The execution time of floating-point instructions also depends on the FPU implementation; the ERC32 and the LEON AT697E have different FPU execution times.

By default, Bound-T uses the worst-case times for the chosen target processor; in the present version only the ERC32 is supported. A command-line option to use instead the "typical" times is provided as well as a command-line option by which the user can specify the floating-point instruction times in a configuration file. The options are described in section 4.2.

2.4 Timing Approximations

The following architectural features can lead to approximate (over-estimated) execution times for the concerned instructions:

- The register window traps.
- The possible concurrent operation of the floating-point unit and the integer unit.
- The data-dependent, variable floating-point execution time.
- The memory wait states that vary for different memory types.

See section 3.14 for more information about the approximations.

2.5 Stack Usage Analysis for the SPARC

An optional Bound-T function is to compute an upper bound on the stack usage of a subprogram and its callees. This analysis is activated by the command-line options *-stack* or *-stack_path*. This analysis is available for all SPARC processors that use the V7, V8 or V8E architectures, including the ERC32 and the LEON.

For the stack-usage analysis, Bound-T first finds the size of the stack frame for each subprogram by analysing the instructions that modify the stack pointer. Next, Bound-T analyses the call graph to find the call path that has the largest total stack usage.

The SPARC architecture defines one stack located in main memory. The stack is used for passing parameters, for local variables, and for spilling register windows on overflow of the register file. Register **r14** is the dedicated stack pointer and register **r30** is the dedicated frame pointer. The standard SPARC calling protocol uses the **SAVE** instruction to allocate a stack frame and the **RESTORE** instruction to deallocate a stack frame. However, a SPARC program can also change the stack and frame pointers with other instructions such as **ADD** and **SUB**.

The stack-usage analysis in Bound-T for the SPARC relies on the standard calling protocol. Currently, the analysis allows at most one **SAVE** instruction per call; this instruction can occur immediately after the **CALL** (in the delay slot) or somewhere in the called subprogram (but not both). The **CALL** (if present) must be matched by a **RESTORE** that is usually the last instruction of a subprogram, in the standard instruction sequence for returning from a subprogram. These rules take special forms in “tail calls” on which more later in section 3.11.

If **SAVE** or **RESTORE** instructions occur in other ways, an error message is emitted and the stack-usage analysis fails.

The program is allowed to use other instructions to modify the stack pointer or frame pointer registers. This may be necessary for subprograms that have a lot of local data or local data of a dynamically computed size, for example a local array with a size defined by an input parameter.

Some programming languages or compilers may use additional stacks under software control. At present, Bound-T has no support for any additional stacks on the SPARC.

3 SUPPORTED SPARC FEATURES

3.1 Overview

This chapter specifies which SPARC instructions, registers and status flags are supported by Bound-T and how Bound-T models them. We will first describe the extent of support in general terms, with exceptions listed later. Note that in addition to the specific limitations concerning the SPARC processor, Bound-T also has generic limitations as described in the Reference Manual [1]. For reference, these are briefly listed in section 3.4.

General support level

In general, when Bound-T is analysing a target program for the ERC32 SPARC processor, it can decode and correctly time all IU instructions, with minor approximations. FPU instructions have more important timing approximations.

Bound-T can construct the control-flow graphs and call-graphs for all instructions when all branches, jumps and calls have *static* target addresses. Some, but not all, forms of dynamic jumps can be analysed. Analysis of dynamic calls (calls via function pointers) is not provided, but the targets of such calls can be defined by assertions.

Bound-T supports the SPARC call/return protocol, using the **CALL** instruction and the and the standard SPARC instruction sequences that return from a subprogram, with or without the register windowing with **SAVE** and **RESTORE** instructions. The alternate call/return protocol that uses an **UNIMP** instruction is also supported.

Bound-T can also analyse trap handler subprograms. Explicit invocation of trap handlers with the **Ticc** instruction is best supported if the trap number is statically available in the trap instruction (immediate operand). Bound-T also tries to analyse trap numbers given as register operands but this is likely to succeed only when the register is given a single static value in the subprogram that contains the **Ticc** instruction.

When analysing loops to find the loop-counter variables, Bound-T is able to track all the 32-bit integer (fixed point) additions and subtractions. Bound-T detects when this integer computation is overridden by other computations, such as multiplications, divisions, logical operations or loads (of any size) into the same registers.

In summary, for a program written in a compiled language such as Ada or C, it is unlikely that the Bound-T user will meet with any constraints or limitations that are specific to the SPARC target system, apart from the approximations in floating-point execution time and possibly in register window trapping.

Before detailing the exceptions to the general support, some terminology needs to be defined concerning the levels of support.

3.2 Levels of Support

Four levels of support can be distinguished, corresponding to the four levels of analysis used by Bound-T:

1. *Instruction decoding*: are all instructions correctly recognised and decoded? Is the execution time of each instruction correctly and exactly included in the WCET, or only approximately?

2. *Control-flow analysis*: are all jump, call and loop instructions correctly traced to their possible destinations? Are there other instructions that could affect control flow, and are they correctly decoded and entered in the control-flow graph?
3. *Definition analysis*: is the effect of each instruction on the data flow correctly traced, in terms of which "cells" (registers, memory locations) are defined (written, modified) by the instruction?
4. *Arithmetic analysis*: to what extent are the arithmetical operations of instructions mastered, so that the range of the results can be bounded?

These levels are hierarchical in the sense that a feature is considered to be supported at one level only if it is also supported at all the lower levels, with arithmetic analysis as the highest level.

Opaque values

When an operation is supported at the definition level, but not at the arithmetic level, then Bound-T's arithmetic analysis considers the operation's results to be unknown or *opaque*.

When an opaque value is stored in a register or memory location, the store is understood to destroy the earlier (possibly non-opaque) value and replace it with the opaque value. For arithmetic analysis, an opaque value represents an undefined value from the set of possible values of the storage cell (32 bits for an IU register, 1 bit for a condition flag).

The difference between definition analysis and arithmetic analysis is crucial to Bound-T's ability to bound the worst-case times of loops. To illustrate this difference, the following table lists some SPARC instructions and their definition-analysis and arithmetic analysis. The instructions are assumed to be executed in sequence. The analysis contains just the aspects supported by Bound-T.

Note that the IU global register **r0** has special semantics: a read of **r0** returns zero, and a write into **r0** has no arithmetic effect. In the assembly code, register number **i** is written as "**%ri**" or just "**%i**". Thus, the instruction in the first row, "**add %r0, 33, %r4**" means to add the value of **r0** (which is zero) to 33 and store the result in **r4**.

Table 1: Definition Analysis vs Arithmetic Analysis

#	*Instruction*	*Definition analysis*	*Arithmetic analysis*
1	**add %r0, 33, %r4**	**r4** gets a new value.	**r4** gets the value 33.
2	**addcc %r4, 1, %r5**	**r5** gets a new value. **Z** and **N** get new values.	**r5** gets the value **r4** + 1, which is 34. **Z** and **N** get the value 0.
3	**subcc %r4, %r5, %r0**	**Z** and **N** get new values.	**N** gets the value 1, since **r4** < **r5**. **Z** gets the value 0.
4	**mulscc %r4, %r5, %r7**	**r7** gets a new value. **Z** and **N** get new values.	**r7** gets an opaque value (Bound-T does not support multiplication in arithmetic analysis). **Z** and **N** get opaque values.
5	**wr %r4, %r5, %y**	No effect. The **Y** register is not tracked.	No effect.
6	**rd %y, %r5**	**r5** gets a new value.	**r5** gets an opaque value (any reading of an untracked register is opaque).

#	*Instruction*	*Definition analysis*	*Arithmetic analysis*
7	**subcc %r5, %r4, %r1**	**r1** gets a new value.	**r1** gets the value **r5 − r4**.
		Z and **N** get new values.	**Z** is set if **r5** = **r4**, otherwise cleared.
			N is set if **r5** < **r4**, otherwise cleared.

Note that in row 7, arithmetic analysis tracks the fact that **r1** is now the difference between **r5** and **r4**, even though **r5** has an opaque value. This tracking is important, for example when Bound-T examines the way a loop-body modifies a variable, to see if the variable is the loop-counter.

In fact, the same holds for all the table rows: arithmetic analysis tracks the formulae, not the values; the values (or value ranges) are then calculated from the formulae when needed.

3.3 Implications of Limited Support

Looking at the support levels from the Bound-T user's point of view, the following implications arise when the target program uses some SPARC feature which is not supported at some level.

- *Arithmetic analysis*: If a feature is supported at all levels except arithmetic analysis, then using this feature in any loop-counter computation will keep Bound-T from identifying the loop-counters (due to opaque values) so these loops cannot be bounded automatically. However, the other results from Bound-T stay valid.

For example, if the initial value of a loop-counter is read from the **Y** register as for **r5** in Table 1, row 6, then Bound-T cannot compute bounds for the initial value and thus cannot bound the loop.

- *Definition analysis*: If a feature is not supported in definition analysis, then in addition to the preceding impact, using this feature implies a risk of invalidating the arithmetic analysis, and thus a risk of incorrect results from Bound-T. Few SPARC features are at this level of non-support, and Bound-T will warn if they are used.

For example, if the instruction "**rd %y, %r5**" in Table 1, row 6 were not supported in the definition analysis, it would not be seen to store a new value in **r5**, and the next instruction in row 7 would seem to get **r5**'s value from row 2, which would be quite wrong.

- *Control-flow analysis*: If a processor feature is not supported in control-flow analysis, then Bound-T can produce arbitrary (correct or incorrect) results when this feature is used in the target program, because the correct control-flow graphs cannot be determined. Again, Bound-T will warn of such usage.
- *Instruction decoding*: If a feature is not supported even for decoding, then it is useless to run Bound-T on a target program that uses this feature, since the only reliable result will be error messages.

3.4 Reminder of Generic Limitations

To help the reader understand which limitations are specific to the SPARC architecture, the following compact list of the generic limitations of Bound-T is presented.

Table 2: Generic Limitations of Bound-T

Generic Limitation	*Remarks for SPARC target*
Understands only integer operations in loop-counter computations.	All results from floating-point operations can be considered opaque.
Understands only addition, subtraction and multiplication by constants, in loop-counter computations.	No implications specific to the SPARC.
Assumes that loop-counter computations never suffer overflow.	The PSR flag **V** is assumed to be zero, except when it is explicitly tested, in which case it is considered unknown (opaque). The **C** flag is considered unknown (opaque).
Can bound only counter-based loops.	No implications specific to the SPARC.
May not resolve aliasing in dynamic memory addressing.	No implications specific to the SPARC.

3.5 Support Synopsis

The following table gives a synoptical view of the level of support for SPARC features. A plus '+' in a cell means that the feature corresponding to the table row is supported on the level corresponding to the table column. A shaded cell indicates lack of support.

The features are ordered from the fully supported at the top, to the unsupported at the bottom. More detail on the support level is given in the following sections.

Table 3: Synopsis of SPARC Support

SPARC registers, instructions, or other features	*Decoding*	*Control flow*	*Definition*	*Arithmetic*	*Remarks*
IU addition, subtraction, **LOAD**, **STORE**, **SETHI**	+	+	+	+	
XOR when both operands are the same register	+	+	+	+	The result is zero.
OR when one operand is literally zero (the **MOV** pseudo-instruction)	+	+	+	+	The result is the other operand.
OR with a literal operand that extends an immediately preceding **SETHI**	+	+	+	+	Equivalent to loading a 32-bit immediate operand into the destination register.
PSR flags **Z**, **N**	+	+	+	+	
IU conditions for signed comparisons, in the **Bicc** and **Ticc** instructions: **BN**, **BE**, **BLE**, **BL**, **BNEG**, **BA**, **BNE**, **BG**, **BGE**, **BPOS**	+	+	+	+	The synonym mnemonics are not listed but are supported since the binary encoding is the same.
CALL followed by an instruction (not **SAVE**) that does not modify the return address	+	+	+	+	The callee may execute **SAVE** to allocate a register window for itself.

SPARC registers, instructions, or other features	*Decoding*	*Control flow*	*Definition*	*Arithmetic*	*Remarks*
CALL followed by **SAVE**	+	+	+	+	Implies that the callee will not itself execute **SAVE** since the caller did it on behalf of the callee.
CALL followed by **RESTORE** when the subprogram is executing in its own register window	+	+	+	+	Tail call; the RESTORE deallocates the subprogram's register window and makes the callee return to the caller of the subprogram.
CALL followed by an instruction (not **RESTORE**) that modifies the return address register	+	+	+	+	If the subprogram is executing in its caller's register window and Bound-T can deduce that the return address is set to the return address of the subprogram itself, this is a tail call. If Bound-T can deduce the static value of the return address, this is an ordinary call (not a tail call) with an unusual return address. Otherwise this is a call with an unknown return address and the control-flow analysis fails.
SAVE alone	+	+	+	+	Only one stand-alone execution of **SAVE** in each subprogram; to allocate the register window for the subprogram; only allowed if the caller does not **SAVE** as part of the call.
SAVE in the delay slot of a **CALL** or of a **JMPL** that implements a call	+	+	+	+	Allocates the register window for the callee; implies that the callee will not itself execute **SAVE**.
UNIMP as part of call sequence ([7])	+	+	+	+	
JMPL in standard return sequences	+	+	+	+	
JMPL as an indirect jump (not a call)	+	+	+	+	Assuming that Bound-T's analysis finds the target(s) of the jump.
Ticc with a static (immediate) trap number	+	+	+	+	Value of TBR must be known.
Ticc with a dynamic (register) trap number	+	+	+	+	The register must be given a single, static value in the subprogram that contains the **Ticc**. Also, the value of TBR must be known.
Store barrier (SPARC V8): **STBAR**	+	+	+	+	The execution time of this instruction may be underestimated for complex memory interfaces.
IU conditions for unsigned comparisons, in the **Bicc** and **Ticc** instructions: **BLEU**, **BLU**, **BGU**, **BGEU**	+	+	+	+	By default an unsigned condition is approximated with the corresponding signed condition. The option *-no_unsigned_cond* makes unsigned conditions opaque.

SPARC registers, instructions, or other features	*Decoding*	*Control flow*	*Definition*	*Arithmetic*	*Remarks*
IU shift, **AND**, **OR**, **XOR** for two different registers	+	+	+		Condition codes that get a constant value are supported in arithmetic. Constant-propagation analysis supports the bit-wise logical operations.
IU **ANDN**, **ORN**, **XORN** (second operand negated)	+	+	+		The result and the condition codes are opaque.
IU multiplication and division (SPARC V8): **UMUL**, **SMUL**, **UDIV**, **SDIV**, **DIVS** (V8E) IU multiply-accumulate (SPARC V8E): **UMAC**, **SMAC**	+	+	+		Opaque values result.
Find leading bit (SPARC V8E): **SCAN**	+	+	+		Opaque values result.
PSR condition codes **V**, **C**.	+	+	+		These condition codes are opaque.
Tagged values: **TADDCC**, **TADDCCTV**, **TSUBCC**, **TSUBCCTV**	+	+	+		Opaque values result in the destination register and the condition codes.
Reading control registers: PSR, WIM, TBR	+	+	+		Opaque values are read.
Operations on the **Y** register	+	+	+		The **Y** register is not modeled as a cell. Writing to it has no arithmetic effect, and opaque values are read from it.
FPU operations on single-precision (32-bit) and double-precision (64-bit) values	+	+	+		FP computation is not modelled.
FPU operations on extended (96-bit) or quadruple (128-bit) values	+	+	+		The execution time is unknown by default. An error message is emitted unless the execution time is defined with the option *-fpu_time*. Disassembled instruction mnemonics use the 'x' suffix for “extended”.
FBfcc condition codes	+	+	+		FP branch conditions are opaque.
Store FPU register to memory: **STF**, **STDF**, **STFSR**, **STDFQ**	+	+	+		If the IU loads the memory location that was stored from the FPU, an opaque value results.
Assignment to processor status register, PSR	+	+	+		Condition codes become opaque.
Ancillary state registers (SPARC V8): **RDASR**, **WRASR**	+	+	+		The ASRs are not modelled as arithmetic cells. Writing an ASR has no effect; reading an ASR returns an opaque value.
Memory access with an Alternate Space code	+	+			
SAVE and **RESTORE** in other contexts	+	+			Warning message will be generated. Control flow analysis may be disturbed if the return address of the executing subprogram is altered.
CALL followed by **RESTORE** when the subprogram is executing in its caller's register window	+				Interpretation to be defined. Error message is emitted.
Assignment to control registers: WIM, TBR	+				The effect on register window traps is not modelled.

SPARC registers, instructions, or other features	*Decoding*	*Control flow*	*Definition*	*Arithmetic*	*Remarks*
Ticc with dynamic trap number that is not resolved to a single static value, or a dynamic call constructed with **JMPL**.	+				Assertions can specify the targets (callees) of a dynamic call, otherwise error message results.
Unimplemented instructions (eg. from SPARC versions after V8E)					Error message is emitted.

3.6 Data Registers and Memory Accesses

The SPARC architecture contains several sets of registers with different roles. This section explains how Bound-T supports these registers.

Integer Unit register file

Bound-T supports all SPARC register file locations that are accessible in one subprogram that follows the standard calling convention. Such a subprogram can access two register windows: before the subprogram executes a **SAVE** instruction it accesses the caller's register window (assuming that the caller did not execute a **SAVE** as part of the calling sequence); after the subprogram executes a **SAVE** instruction it accesses its own register window. Each “out”, “local” and “in” register in these two register windows is modelled as a separate data cell, but taking into account the overlap between the windows which means that the “out” registers of the caller's window coincide with the “in” registers of the callee's window. The storage cells that modell these coinciding “out” and “in” registers are called “pass” register cells. To distinguish between the “local” registers in the caller's and callee's windows, the storage cells that model the callee's “local” registers are called “work” register cells. More on this system of storage cells in section 5.3.

Of course there is only one set of “global” registers and storage cells that model them, not connected to any register window.

Bound-T assumes that **r14** is used as the stack pointer and **r30** as the frame pointer, as is normal in SPARC programs. Section 3.7 explains this further.

The **Y** register is not modelled. Its value is considered unknown (opaque).

Floating-point Unit registers

Floating-point operations are not supported in arithmetic analysis (because loop counters are seldom floating-point variables). The FPU registers **f0** - **f31** are not modelled as data cells. A store from the FPU to memory is modelled as writing an opaque value into the memory word, which may later enter the IU calculations through an IU load from this memory word.

Memory data

Bound-T generally models SPARC memory locations as 32-bit words. Thus, when a load instruction reads a word from memory into a register, Bound-T models it as a 32-bit assignment (not opaque). The result of instructions that load a byte, a half-word or a

double-word is considered opaque (unknown). However, for byte and half-word loads the arithmetic analysis uses the range bounds that the data-type places on the loaded value. The following table explains the modelling of memory loads.

Table 4: Modelling Load Instructions

Instruction	*Data-type loaded*	*Loaded value*	*Range*
LDUB, LDSTUB	unsigned 8 bits	opaque	0 .. 255
LDSB	signed 8 bits	opaque	-128 .. 127
LDUH	unsigned 16 bits	opaque	0 .. 65 535
LDSH	signed 16 bits	opaque	-32 768 .. 32 767
LD	signed 32 bits	the value of the memory location	not applied
LDD	64 bits	opaque	not applied

The range bounds applied to byte and half-word loads may give loop bounds for loops that use byte or half-word counters, even if Bound-T is unable to analyse the loop termination condition. Of course such bounds may be very overestimated.

Range bounds are not applied to word or double-word loads because they could give ridiculously overestimated loop bounds.

When an instruction stores a 32-bit word from an IU register into memory, Bound-T models the instruction as a 32-bit assignment to the memory word (not opaque). Instructions that store smaller data types (octets or half-words) are considered to store an opaque value in the destination word. Instructions that store doublewords are considered to store opaque values into both destination words.

For references to stack data see section 3.7.

When a store or load instruction uses an “alternate address space identifier” (ASI) Bound-T ignores the effect of the store and considers the load to give an opaque value.

3.7 Registers and the Calling Protocol

Windowed or flat

Some compilers that target the SPARC support a “flat” register model as an alternative to the standard “windowed”. The “flat” model uses only one set of IU registers and never uses **SAVE** or **RESTORE** instructions. Bound-T was designed to support the “windowed” model but should be able to analyse “flat” code, although some spurious warnings may be emitted because Bound-T then classifies all subprograms as “leaf” subprograms that should not make further calls. With the “flat” model the register-window trap analysis is unnecessary and should not be used (do not use the option *-rw*).

Stack and frame pointers

The registers that are used by the standard SPARC calling protocol are **r31** for the return address, **r30** for the frame pointer, **r14** for the stack pointer, and **r15** as a return address from a “leaf” subprogram. The standard calling protocol assumes that the values of these registers are preserved between corresponding call and return. A subprogram that does not

follow this calling protocol (such as a task-switching kernel routine) should be analysed separately, because Bound-T is unable to follow the control-flow into and out of such a subprogram.

If a subprogram manipulates the calling-protocol registers in some other way, Bound-T generally emits an error message. However, in some cases modification of calling protocol registers is necessary. For example, the register underflow trap handler copies the register values from memory (stack) to the register set, including the frame pointer register. When Bound-T is analysing this trap handler it therefore allows instructions that modify the frame pointer and the error message is suppressed.

References to stack variables

SPARC memory references in load and store instructions typically compute the memory address as the value of a base register plus an offset, where the offset can be a static value (immediate operand) or a dynamic value from a register. Bound-T interprets memory references based on the stack pointer **sp** = **r14** or the frame pointer **fp** = **r30** as references to local variables or to stacked parameters, depending on which register is used and on the sign of the offset. The details are shown in the table below.

Table 5: References to Stack Variables

Base	*Offset*	*Interpretation*
sp = **r14**	≥ 0	A stack location. When the subprogram is using its own register window this is usually a location that passes a parameter from this subprogram to a callee subprogram (an "out" location), otherwise (caller's register window) this is a location that passes a parameter to the current subprogram from its caller.
	< 0	An illegal reference, because the location is outside the allocated stack and is therefore volatile on interrupts. A warning is emitted.
fp = **r30**	≥ 0	A stack location that passes a parameter to this subprogram from its caller subprogram (an "in" location).
	< 0	A local variable in this subprogram.

3.8 Condition Codes

The IU condition codes that are modelled in arithmetic analysis are **Z** (result zero) and **N** (result negative). The **V** condition code (overflow) is assumed to be always zero in possible loop conditions, because of the generic limitations in Bound-T, and the definition of the **N** condition code is simplified to the negativity of the result of the arithmetic operations. See Table 6 below, where O1 and O2 represent the operands of the operation.

Table 6: Definition of N Condition Code from Arithmetic Operation

Operation	*Condition*
Addition, result = O1 + O2	**N** is set if O1 + O2 < 0.
Subtraction, result = O1 - O2	**N** is set if O1 - O2 < 0.

The SPARC architecture defines different branch conditions, in the **Bicc** and **Ticc** instructions, for *signed* comparisons and for *unsigned* comparisons. Bound-T for the SPARC generally models the arithmetic as signed, therefore it should consistently model the unsigned conditions as opaque. However, compilers frequently use the unsigned comparisons for loop counting, so by default Bound-T approximates each unsigned condition by the corresponding signed condition. This approximation can be disabled with the option *-no_unsigned_cond* to make the arithmetic analysis safer but also weaker.

The following table shows the model of the SPARC branch conditions (see [8]) under default options and under *-no_unsigned_cond*.

Table 7: Modelling the SPARC Condition Codes

Mnemonic	*Condition*	*Arithmetic model*	
		Default	*-no_unsigned_cond*
BN	Branch Never	*false*	*false*
BE	Branch on Equal	**Z**	**Z**
BLE	Branch on Less or Equal	**Z** or **N**	**Z** or **N**
BL	Branch on Less	**N**	**N**
BLEU	Branch on Less or Equal, Unsigned	**Z** or **N**	opaque
BLU	Branch on Less, Unsigned	**N**	opaque
BNEG	Branch on Negative	**N**	**N**
BVS	Branch on oVerflow Set	opaque	opaque
BA	Branch Always	*true*	*true*
BNE	Branch on Not Equal	not **Z**	not **Z**
BG	Branch on Greater	not (**Z** or **N**)	not (**Z** or **N**)
BGE	Branch on Greater or Equal	not **N**	not **N**
BGU	Branch on Greater, Unsigned	not (**Z** or **N**)	opaque
BGEU	Branch on Greater or Equal, Unsigned	not **N**	opaque
BPOS	Branch on Positive	not **N**	not **N**
BVC	Branch on oVerflow Clear	opaque	opaque

Bound-T cannot automatically bound loops that use opaque branch conditions. Instead user assertions are required to bound the repetition count of the loop.

Note that **BVS** and **BVC** are considered opaque, even if Bound-T generally assumes that arithmetic operations do not overflow and so **V** should be zero. If the program under analysis uses such branch conditions, it must be assumed that the value of **V** can change.

Direct assignment to the PSR register via the WRPSR instruction is modelled as storing opaque values in the condition codes.

Control/status registers other than PSR (WIM, TBR, **Y**) are not modelled as arithmetic cells. Writing one of these control/status registers has no effect on the analysis; reading a control/status register returns an opaque value.

The FPU condition codes are not modelled at all; all conditions are considered opaque in the **FBfcc** instructions.

3.9 Computational Operations

Whether or not a computational operation is supported on the arithmetic analysis level depends exclusively on the generic abilities of Bound-T; the only concern here is to map these abilities onto the SPARC instruction set.

Supported Integer Unit arithmetic

All IU operations are supported for definition analysis. The following operations are supported for arithmetic analysis:

- **ADD**, **ADDCC**
- **SUB**, **SUBCC**
- **XOR**, **XORCC** when the source registers are the same, giving zero.
- **OR**, **ORCC** when one operand is known to be zero, giving the value of the other operand (this is the **MOV** pseudo-instruction).
- **OR**, **ORCC** when the dynamically preceding instruction is **SETHI** and the **SETHI** destination register equals the first **OR** source operand and the second **OR** source operand is an immediate (static) constant. In this case, the **SETHI** and **OR** together load an immediate 32-bit value into the **OR** destination register.
- **SLL** when the shift count c is an immediate operand small enough to model the left-shift as a multiplication of the source register by 2^c. The option *-sll_max* sets the limit on shift count (default $c \leq 10$).
- **SLL** immediately followed by **SRA** when the two instructions can be combined into a "masking" operation where the **SLL** destination register equals the **SRA** source operand and both instructions have the same immediate shift count operand.

For these operations, the arithmetic effect is supported for the PSR condition codes **Z** and **N**. Furthermore, Bound-T's constant-propagation analysis phase supports the bitwise logical operations (**AND**, **OR**, **XOR**) fully, not just in the special cases defined above.

When programming in assembly language, it is advisable to limit all loop-counter arithmetic to use only the above operations (and other features supported on the arithmetic level). This will maximise Bound-T's automatic loop-bounding ability.

Immediate operands

Immediate (literal) operands are considered as unsigned numbers for the **AND**, **OR** and **XOR** operations and as signed numbers for all other supported integer operations.

The SPARC instruction set limits the size of immediate operands. To load a 32-bit immediate value into a register programs use a **SETHI** instruction to load the high bits followed by an **OR** instruction with an immediate operand that defines the low bits. Bound-T detects such instruction pairs and models the full 32-bit constant.

Unsupported Integer Unit operations

The following IU operations are not supported in arithmetic analysis, except for the special cases described above:

- **ADDX**, **ADDXCC**
- **SUBX**, **SUBXCC**

- **MULSCC**
- **UMUL**, **SMUL**, **UDIV**, **SDIV**, **DIVS**, **UMAC**, **SMAC**
- **AND**, **ANDCC**
- **ANDN**, **ANDNCC**
- **OR**, **ORCC**
- **ORN**, **ORNCC**
- **XOR**, **XORCC**
- **XNOR**, **XNORCC**
- **SLL**, **SRL**, **SRA**.

In arithmetic analysis these operations are understood to store opaque values in the target register and the condition codes. However, if an operation yields a constant condition code value (usually zero), then this condition code is supported arithmetically for this operation.

Floating-point operations

FPU operations on 32-bit and 64-bit values are decoded and their execution time is modelled, but not their arithmetic. FPU operations on 96-bit or 128-bit values are decoded but have no default execution time so an error message is emitted unless the execution time is defined with the option *-fpu_time*. The arithmetic of 96- or 128-bit floating-point operations is also not modelled.

The result of any FPU comparison is opaque; the condition on an FPU-related conditional branch is considered unknown.

The only interaction between the FPU and the arithmetic or definitional analysis occurs when an FPU store instruction assigns a value to a memory location; if the (detectably) same location is then loaded into an IU register, an opaque value is assumed.

Example of the unsupported operations

To illustrate the effect of the unsupported instructions on the analysis, the following table lists some supported and unsupported SPARC instructions with the arithmetic analysis and explains the effect on the analysis. The instructions are assumed to be executed in sequence.

Table 8: Effect of Unsupported Instructions on the Analysis

#	*Instruction*	*Arithmetic analysis*	*Effect on the analysis.*
1	**or %r0, 10, %r1**	**r1** gets the value 10.	The value of **r1** is known.
2	**sll %r1, 3, %r2**	**r2** gets the value **r1** * 8, which is 80.	The value of **r2** is known.
3	**andcc %r1, %r2, %r0**	Condition codes **Z** and **N** get opaque values.	The next branch (**bneg**) has an opaque condition.
4	**bneg xxx**	No arithmetic effect.	Opaque branch condition. If this is a loop branch the loop cannot be bounded automatically.
5	**sra %r2, 1, %r2**	**r2** gets an opaque value.	If **r2** is involved in counting loops, those loops cannot be bounded automatically.

#	*Instruction*	*Arithmetic analysis*	*Effect on the analysis.*
6	**mulscc %r1, %r2, %r3**	**r3**, **Z** and **N** get opaque values.	If **r3** is involved in counting loops, those loops cannot be bounded automatically.
7	**bcs xxx**	The condition is opaque. No arithmetic effect.	Opaque branch condition. If this is a loop branch the loop cannot be bounded automatically.

As can be seen from Table 8, if the program counts loop iterations by means of operations that arithmetic analysis does not support, Bound-T will not be able to bound the loop automatically. User assertions are required for such loops.

3.10 Division and Remainder Routines

The ERC32 (SPARC V7 architecture) has no integer division instructions so compilers provide library functions for this purpose, usually identical or similar to the functions recommended in the SPARC manual (Program 6 in [9]). Unfortunately the recommended function is written in a form that makes the control-flow graph "irreducible" which means that Bound-T cannot find a WCET bound. The WCET of these functions must be determined in some other way and supplied to Bound-T with assertions.

We have measured the execution time of these functions in the GNU "libc" library on an ERC32 for a large number of randomly generated parameters. The assertions quoted below show the maximum execution time we measured. Of course, there is no guarantee that this is actually the WCET for these functions, moreover your library may have different versions of these functions (although this seems unlikely).

```
subprogram ".div"
   time 149 cycles;
end subprogram ".div";

subprogram ".udiv"
   time 150 cycles;
end subprogram ".udiv";

subprogram ".rem"
   time 156 cycles;
end subprogram ".rem";

subprogram ".urem"
   time 154 cycles;
end subprogram ".urem";
```

3.11 Jumps, Calls, Returns and Traps

Branch and jump instructions

The branch instruction **Bicc** using IU conditions is supported on all levels. Arithmetic analysis does not support the floating-point conditions (**FBfcc**) nor the coprocessor conditions (**CBccc**); those conditions are considered opaque.

The jump-and-link instruction **JMPL** is supported on all levels. However, there are generic limitations (see [1]) on the control-flow analysis of indirect jumps, that is, **JMPL** where the operands that define the target address are registers with dynamically computed values.

Jump-and-link can also be used as a register-indirect call instruction (see below) and as a part of the SPARC "control-transfer couples" (also see below). **JMPL** instructions that do not have these standard forms may not be supported on the control-flow level; warnings or errors are emitted for them.

Jump via address table

For dense switch/case statements some compilers generate code that uses a table of addresses to jump to the correct case-branch. This code first converts the switch/case index expression to the address of a table slot (element), then loads the address from this table slot into a register, and finally jumps to the address in this register. The last two instructions are thus of the form

```
LD      [ x ], r
JMPL    [ r ], r0
```

where *x* is the expression (base + offset) for the address of the table slot and *r* is a working register that holds the address loaded from the table. Bound-T detects this code idiom, called "jump via table", and analyses the **LD** and **JMPL** instructions together. Bound-T uses arithmetic analysis to find bounds on *x*. These bounds show the address and length of the table, so Bound-T can read out the contents of the table (the addresses of the case-branches) which gives the possible targets of the **JMPL** and thus defines the continuation of the control-flow graph after the **JMPL**.

Call instructions

The **CALL** instruction is supported on all levels. A **JMPL** with destination degister **r15** is equivalent to a **CALL**, but can have a dynamically computed target address (callee entry address); there are generic limitations (see [1]) on the control-flow analysis of such dynamic calls but you can define the possible target addresses with assertions.

The SPARC call/return protocol varies in three ways:

- whether **SAVE** and **RESTORE** instructions are used to provide the called subprogram with its own register window,
- whether an **UNIMP** instruction is used to supply an additional immediate parameter to the called subprogram (according to the manuals, this parameter defines the size of the structure that the called subprogram should return),
- whether the subprogram is a trap handler.

Leaf vs non-leaf subprograms

A subprogram that uses the same register window as its caller is known as a *leaf* subprogram because it usually contains no calls itself and so is a leaf node in the call tree. A subprogram for which a register window is allocated with **SAVE** is boringly called a *non-leaf* subprogram.

Bound-T classifies a subprogram as a non-leaf subprogram if the subprogram executes a **SAVE** instruction (we call this a *self-save* subprogram) or if all calls to the subprogram have a **SAVE** instruction as the delay instruction (we call this a *caller-save* subprogram).

Bound-T classifies a subprogram as a leaf subprogram if the subprogram does not execute a **SAVE** instruction and the calls to the subprogram do not have **SAVE** instructions either (as the delay instruction of the **CALL** instruction).

Bound-T checks that the presence and placement of the **SAVE** instruction is consistent for all calls. For example, if a self-save subprogram is called with a caller-save sequence (a **CALL** followed by **SAVE**) an error message ("mismatch of RW kind") results.

Normal vs alternate calls

Calls and returns that do not use **UNIMP** are called *normal* calls and returns, while those that use **UNIMP** are called *alternate* calls and returns.

The *normal* calling sequence consists just of a **CALL** or **JMPL** instruction and its delay instruction (perhaps a **SAVE**). The callee returns to the instruction following the delay instruction, at (address of **CALL**) + 8, unless the delay instruction loads another return address into **r15**. in which case the callee returns to **r15** + 8.

The *alternate* calling sequence consists of a **CALL** or **JMPL** instruction and its delay instruction as above, and one following **UNIMP** instruction at the normal return point, (address of **CALL**) + 8, or at the return address defined by loading **r15** in the delay slot, **r15** + 8. The callee returns to the instruction following the **UNIMP**, at (address of **CALL**) + 12 or **r15** + 12.

Bound-T checks for the **UNIMP** instruction and chooses the assumed return point accordingly. However, Bound-T does not model the way in which the callee might access the additional parameter from the **UNIMP** operand.

In principle, the called subprogram (the callee) could also check its own return address and adapt to being called with either calling sequence, normal or alternate. However, Bound-T assumes that a given subprogram always uses the same sequence, and therefore it checks that all calls of a given subprogram use the same form of calling sequence. Bound-T also checks that the callee uses the return sequence (normal or alternate return) that corresponds to its calling sequence (normal or alternate call).

The **UNIMP** instruction is supported only as a part of the alternate calling sequence. Otherwise, this instruction causes a warning message to be emitted. The instruction is then modelled as a return from the subprogram (a stop-gap to let the analysis continue).

Tail calls

When the last action in a subprogram is to call another subprogram, a compiler can sometimes optimize the call-return sequence by using a **CALL** instruction immediately followed (in the delay slot) by some instruction that removes the subprogram itself from the call chain and makes the callee return to a higher level, bypassing the subprogram that executes this **CALL**. The term *tail call* is used for such constructs.

Bound-T detects and understands two forms of tail calls:

- If the subprogram is executing in its own register window, a **CALL** immediately followed by a **RESTORE** instruction in the delay slot is a tail call. The **RESTORE** instruction discards the register window and stack frame of the caller, before the callee is entered, so that the callee can reuse this part of the register file and will return directly to the caller's caller (or even higher in the call chain, if there were tail calls on preceding levels).
- If the subprogram is not executing in its own register window, a **CALL** immediately followed by an instruction that sets the return address register **r15** to the value it had on entry to this subprogram is a tail call. Register **r15** defines the return address for the callee, thus the call will return to caller's caller (or even higher in the call chain).

When Bound-T detects such tail calls it models them correctly in the control-flow analysis. (The analysis of tail calls in the register window trap analysis is currently under review and to be confirmed.) However, the second form of tail call is detected only if the instructions that carry the return address from the entry point of the subprogram to the delay-slot instruction are simple enough to be analysed as "copy" operations (as defined for the "value-origin" analysis, see [1]).

Trap handler subprograms

Bound-T classifies a subprogram as a trap handler if it is called with the **Ticc** instruction or if it ends with one of the two standard return-from-trap control transfer couples (see discussion of return instructions below).

Traps and trap instructions

The explicit trap instruction, **Ticc**, with a static trap number, is modelled as a conditional call to the corresponding entry in the trap vector. If the trap number is specified dynamically by a register operand, Bound-T tries to bound the value of the register (using constant propagation and arithmetic analysis); if the result is a single value, the instruction is modelled as a conditional call as for a static trap number, otherwise the instruction is considered a dynamic call, an error message is emitted and the execution of the trap handler is not modelled. For such unresolved dynamic calls, you can define the possible target addresses with assertions.

The entry address is computed from the trap number and the trap base address as set by the *-trap_base* option. The trap base addresses for a SPARC program is usually determined by the boot code that sets the Trap Base Register (TBR). The entry address for trap number *N* is TBR + 16*N*. For example, if the trap vector is located at the address 2000000 hex, the entry point of trap number 6 (the register window underflow trap) is at 2000060 hex.

The trap vector allocates only 16 octets (4 instructions) to each trap. Therefore, the trap vector usually contains only a jump to the actual trap handler subprogram. The entry address of the trap vector entry is seldom connected to a symbolic identifier which means that Bound-T will not know the true identifier (name) of the trap handler subprogram, but will instead use the hexadecimal address of the trap vector entry as the identifier. For example, if the trap base register is 2000000 hex, the register window underflow trap handler will have the "identifier" 2000060. This identifier will appear in the analysis results for this trap handler.

Many SPARC instructions cause implicit traps in error situations. However, Bound-T includes in the WCET bound only the traps caused by the trap instruction (**Ticc**) and its estimate of the register window traps caused by the **SAVE** and **RESTORE** instructions, as explained in section 2.2. Any other possible trap is not included in the WCET bounds from Bound-T.

Return instructions

All return instructions are supported on all levels. Some restrictions, however, apply to the use of **RETT** (return from trap) instruction, see [8] or [9]. A trap handler subprogram returns with a delayed control transfer couple that consist of a **JMPL** followed by a **RETT**.

The following delayed control transfer couples can be used in trap handlers:

```
JMPL %17, %0
RETT %18
```

to re-execute the trapping instruction, and

```
JMPL %18, %0
RETT %18 + 4
```

to return to the instruction after the trapping instruction.

RETT instructions outside the delayed control transfer couples presented above cause an error message to be emitted.

3.12 SAVE and RESTORE Instructions

SAVE and **RESTORE** instructions are fully supported, but some restrictions apply to their usage in normal subprograms. There are no restrictions on their usage in trap routines.

A normal subprogram that is called with a **CALL; SAVE** (or **JMPL; SAVE**) sequence cannot contain any **SAVE** instruction itself because the **SAVE** in the delay-slot of the **CALL** (or **JMPL**) already creates a register window for the called subprogram. These subprograms are known as *caller-save* subprograms.

A normal subprogram that is called without a **SAVE** in the delay-slot may contain at most one **SAVE** instruction. This **SAVE** instruction may or may not be the first instruction in the subprogram. If there is a **SAVE** instruction the subprogram is known as a *self-save* subprogram, otherwise it is known as a *leaf* subprogram.

A normal caller-save or self-save subprogram must contain one **RESTORE** instruction. This **RESTORE** instruction may be the last instruction in the subprogram, in which case it is the delayed instruction of a return instruction or a tail call, or it may lie earlier in the body of the subprogram. For a self-save subprogram the **SAVE** must come before the **RESTORE** in any execution path.

Executing a **SAVE** instruction can cause a register-file overflow trap, and executing a **RESTORE** instruction can cause a register-file underflow trap. Section 2.2 explains how Bound-T statically predicts the occurrence of these traps and includes their execution time in the WCET bounds.

3.13 Control/Status Registers

The instructions that read IU control/status registers are supported in Bound-T on the definition level but all values read are considered opaque. These registers are the following:

PSR	Program Status Register
WIM	Window Invalid Mask
TBR	Trap Base Register
Y	Multiply Step

Writing values into PSR, WIM or TBR may alter the control flow in a way that Bound-T does not model, and so such writes are supported only on the instruction decoding level. If they occur in the target program, it is the user's responsibility to judge if the results from Bound-T are still valid for WCET analysis.

3.14 Time Approximations

The following table lists the cases where Bound-T uses an approximate model of the timing of SPARC instructions.

Table 9: Approximations for Instruction Times

Case	*Description*	*Maximum Error*
Register window traps	The static analysis may predict more traps than actually can occur because Bound-T assumes worst-case call depths of the subprogram's callees, and the worst initial register window usage of its calls. The prediction of an overflow trap when calling a subprogram *C* from a subprogram *S* does not make use of the specific context in which *S* was called. Likewise, the prediction of an underflow trap on return from a subprogram *C* does not make use of the context in which *C* was called. Thus the register window analysis is independent of the call-path, unlike the WCET analysis which can depend on the call-path.	One trap time per every call and return
If kernel restores only one register window on task resumption	It may be necessary to ask Bound-T to assume that every return causes an underflow trap; see section 2.2.	One trap per every return
Run-time traps other than `Ticc` and other than register window traps	Bound-T does not attempt to analyse when such traps (errors) can occur and does not include the trap handling in the WCET analysis nor in the stack usage analysis.	Unlimited, depending on the trap handlers.
Concurrent operation of FPU and IU	If the IU and FPU of the SPARC operate concurrently, at some points of execution the IU must wait for the FPU to complete its current work. Bound-T tries to optimize the insertion of the additional delays by distributing them into the edges of the control-flow graph. The delays can therefore delay other paths as well in addition to the paths that really need the delays. At worst, the resulting WCET bound may correspond to non-concurrent (sequential) IU/FPU operation in which the IU waits for the FPU to complete each instruction before starting the next IU instruction. However, the worst-case execution path is likely to go through the FP instructions that cause these delays, therefore the error here should be minimal.	The worst-case execution time of the FP instructions
The variable FP execution time	The variable and data-dependent execution time of the floating-point instructions means that the actual execution time of the instruction cannot be statically known and Bound-T must assume the worst-case execution time of the instruction. The worst-case execution times of the floating-point instructions occur usually only when the input values of the instruction are denormalized numbers and therefore the worst-case time is usually pessimistic.	The difference between the FP instruction's actual and worst-case execution times

Case	*Description*	*Maximum Error*
No default execution time for extended or quadruple FP precision	The user must define the execution time with the option *-fpu_times*.	Depends on user-given values.
The varying memory types with different speeds	In typical SPARC implementations there are several types of memories with different speeds and even the memory wait states may differ with the same type of memory. Bound-T assumes the same speed for all memory accesses, but the user may set different memory speed for example for memory accesses in one subprogram. The data and code can lie in different memories and therefore the number of wait states can be given separately for data accesses and for code fetches. See sections 4.2 and 5.6 for details.	Depends on user-given values
Cache memories	At present Bound-T does not model cache memories and must therefore assume that every memory access is a cache miss.	The difference between a cache hit and a cache miss, for every access to memory
Undocumented instruction times with some memory wait states.	On the ERC32 and according to ref. [12] when memory wait states are in use the delay caused by FP instructions to the IU, as well as the WCETs of the instructions STB, STH and LDSTUB do not seem to behave as described in the SPARC documents [4], [5], [6] and [7]. Bound-T uses overestimated instruction times to avoid too low WCET bounds.	Some cycles per FP delay or a critical instruction

4 USING BOUND-T FOR SPARC

This chapter explains how to use Bound-T to analyse SPARC programs. It describes the input file formats (the executable program file and other input files), the command-line syntax, the command-line options, the outputs and the warning and error messages. A dedicated section (4.3) discusses the "HRT mode" of analysis.

We concentrate on information specific to the SPARC version of Bound-T; please refer to the generic Bound-T Reference Manuals [1] [2] [13] for information on the generic inputs, options and outputs, especially for the generic syntax and meaning of the Bound-T assertion language. The last section of this chapter advises on SPARC-specific aspects of the assertion language, for example how to place assertions on the values of SPARC registers.

4.1 Input Formats

Executable file

The target program executable file must be supplied in ELF-32 format. The byte-order should be big-endian, since that is the byte-order of the SPARC. When possible the file should contain symbolic debugging information in DWARF, STABS or ELF form.

File of FP operation times

The option *-fpu_time=filename* tells Bound-T to read the assumed times of FP operations from the file named *filename*. This file must be a text file with line terminators valid for the platform on which Bound-T is run. Blank lines are ignored. Each non-blank line in the file should contain two items: (1) the mnemonic identifier of an SPARC FP operation and (2) a decimal integer giving the number of cycles to be used for this operation. These should be separated by one or several spaces (blanks). For example, the following line specifies 12 cycles for a single-precision addition:

```
FADDS 12
```

The mnemonics are case-insensitive, thus `fadds` and `FADDS` are equivalent. The upper-case forms are the following, in alphabetic order. Note that some forms may not be implemented in the SPARC instruction set; setting the execution time of such a form has no effect on the analysis and is silently accepted (no error or warning message).

FABSS	FDIVD	FMOVS	FSQRTD
FADDD	FDIVS	FMULD	FSQRTS
FADDS	FDIVX	FMULS	FSQRTX
FADDX	FDMULX	FMULX	FSTOD
FCMPD	FDTOI	FNEGS	FSTOI
FCMPED	FDTOS	FXTOD	FSTOX
FCMPES	FDTOX	FXTOI	FSUBD
FCMPEX	FITOD	FXTOS	FSUBS
FCMPS	FITOS	FSMULD	FSUBX
FCMPX	FITOX		

The file can define the times for any subset of FP operations, in any order. The times for other FP operations are not changed. The time given for an FP operation sets both the "typical" and the worst-case time of this operation (to the same value).

The mnemonics with "X" stand for extended-precision or quadruple-precision instructions. Which precision actually is used depends on the SPARC device; the binary instruction encoding is the same. Bound-T has no default execution time for these instructions and will emit error messages if the target program contains such instructions for which an execution time has not been defined with the option *-fpu_times*.

Patch file

Sometimes it is useful to slightly modify or "patch" the target program before analysis. Bound-T provides the general option *-patch=filename* that names a file that contains patches to be applied to the loaded program memory image before analysis starts. The format of the patch file is specific to the target processor. This section explains the patch file format for the SPARC.

The patch file must be a text file with line terminators valid for the platform on which Bound-T is run. Blank and null lines are ignored. Leading and trailing whitespace on each line is ignored. Lines that start with "--" (possibly with leading whitespace) are ignored (as comments).

The remaining lines are patch lines. Each patch line contains two or more fields (tokens) separated by whitespace. The first field is a SPARC address in hexadecimal form and defines the location that is patched; the remaining fields define the data for the patch. The address must be 32-bit aligned (a multiple of 4). The addressed location must be present in some code or data segment loaded from the executable file. In other words, patches cannot be used to extend the loaded memory image, only to change its content.

The table below explains the form and meaning of the patch lines for the SPARC. Two forms are possible, corresponding to the two rows in the table.

Table 10: Patch Formats

Field 1	*Field 2*	*Field 3*	*Meaning*
Address (hex)	32-bit word (hex)		Places the word (field 2) at the patch address (field 1), overwriting the word loaded from the executable file at the patch address.
Address (hex)	"trap"	Target address (hex) or subprogram name	Builds a jump to the target address (field 3). The jump consists of two instructions: A **SETHI** that is placed at the patch address (field 1) and a **JMPL** that follows (at field 1 + 4 octets). These instructions overwrite the words loaded from the executable file at these addresses.

Note that in the second form (second row of the table) field 2 shall contain the literal text "trap" but *without* any enclosing quotes. This form is intended for changing entries in the trap vector.

The hexadecimal values can contain underlines (_) to separate digit groups.

Here is an example of a patch file:

```
-- This is a comment.
-- The following patch line places the instruction
--
--      rd %psr,%l0
--
-- which is "a1_48_00_00" in hexadecimal, at the
-- address 2000810, also in hex:


2000810 a1_48_00_00


-- The following patch line places the two instructions
--
--      sethi %hi(Handler), %l4
--      jmp %l4 + %lo(Handler), %g2
--
-- at address 2000814, where "Handler" is assumed to be
-- a subprogram name, present in the symbol-table:


2000814 trap Handler
```

Note that a comment cannot be appended to a patch line, so the following patch line is wrong:

```
2000810 a1_48_00_00  -- This kind of comment is not allowed.
```

4.2 Command Arguments and Options

The generic Bound-T command format, options and arguments are explained in the Reference Manual [1] and apply without modification to the SPARC version of Bound-T. The command line usually has the form

`boundt_sparc` *options executable-file root-subprogram-names*

For example, to analyse the execution time on the ERC32 processor of the *main* subprogram in the ELF executable file *prog.elf* under the option *-rw*, the command line is

```
boundt_sparc  -device=erc32  -rw  prog.elf  main
```

Root subprograms can be named by the link identifier, if present in the program symbol-table, or by the entry address in hexadecimal form. Thus, if the entry address of the *main* subprogram is 20004A0 (hex), the above command can also be given as

```
boundt_sparc  -device=erc32  -rw  prog.elf  20004A0
```

All the generic Bound-T options apply. There are additional SPARC-specific options as explained below. The generic option *-help* makes Bound-T list all its options, including the target-specific options.

The explanation of the SPARC-specific options is grouped below as follows:

- Target device selection options
- Device-specific options
- Program loading options
- Instruction modelling options
- Register window analysis options
- Floating-Point Unit analysis options
- Memory timing options
- SPARC-specific items for the generic *-trace* option.

There are also options for RapiTime export. See section 4.7 for these.

Target device selection options

You must tell Bound-T which kind of SPARC processor the target program is meant for so that Bound-T can use the right SPARC version and suitable defaults for the trap base address and other parameters.

Use the option *-device=name* to selec the target processor by name. The supported devices, their names for the *-device* option and their properties are listed in the following table, one row per device.

Table 11: Device selection options

Option	*SPARC device*	*Version*	*-trap_base*	*-code_base*	*FPU*
-device=erc32	The ERC32.	V7	2_000_000	2_000_000	Concurrent
-device=v8	The ERC32 extended with the V8 instruction set.	V8	2_000_000	2_000_000	Concurrent
-device=v8e	The ERC32 extended with the V8E instruction set.	V8E	2_000_000	2_000_000	Concurrent

The columns in this table have the following meaning:

- Option: The option that selects the device.
- SPARC device: Identifies the device.
- Version: The version of the SPARC architecture (instruction set) that this device implements.
- *-trap_base*: The default value of the Trap Base Address for this device in hexadecimal form. To override this default value give the option *-trap_base=address* after the *-device* option.
- *-code_base*: The default value of the *-code_base* option for this device in hexadecimal form. To override this default value give the option *-code_base=address* **after** the *-device* option.
- FPU: Whether the FPU executes sequentially or concurrently with respect to the IU. To override this default give the option *-fpu=sequential* or *-fpu=concurrent* **after** the *-device* option.

The *-device=name* option can also be abbreviated to *-name*, for example *-erc32*, unless the name of the selected device happens to equal the name of some other option, which is not the case for the currently supported devices.

If you want to analyse LEON code for control-flow or stack-usage, use the *-v8* or *-v8e* device and the other options *-trap_base=40_000_000* (or whatever Trap Base Address is defined in your LEON program) and *-code_base=40_000_000*. No "leon" device-name is provided because timing analysis is currently not supported for the LEON family.

Device-specific options

For some SPARC devices, Bound-T may require or allow additional options specific to this device. If such device-specific options are used they must be given on the command-line after the *-device* option that selects the device.

At present, there are no device-specific options.

Program loading options

The following table describes the options that control the process of reading the target program from an executable file. The main issue is how to use the symbol-tables (debugging information) that may be in the executable file. Bound-T uses the symbol-tables to map machine addresses to source-level subprogram or variable names and source-file names and line-numbers. Cross-compilers for the SPARC typically generate symbol-tables in one or more of three forms:

– As an ELF symbol-table, a form defined in the ELF standard itself.

– As a STABS symbol-table, a form that predates the ELF standard but can be presented as an ELF section with a specific name.

– As a DWARF symbol-table, the newest and most complete form that can be presented as as a set of ELF sections with specific names.

By default Bound-T uses the DWARF symbol-table if it exists and otherwise the STABS symbol-table if it exists. In both cases, if the ELF symbol-table also exists, Bound-T complements the DWARF or STABS information by also using the "global" ELF symbols. If the executable file contains neither DWARF nor STABS symbol-tables Bound-T uses all of the ELF symbol-table if it exists (as it usually does), taking both "global" and "local" symbols.

Table 12: Program Loading Options for SPARC

Option		*Meaning and default value*
-elf_symbols	*Function*	Makes Bound-T use the whole ELF symbol-table even if DWARF or STABS symbol-tables exist. Depending on the compiler, this may make more subprogram identifiers available.
	Default	Only the "global" ELF symbols are used if DWARF or STABS tables exist.
-stabs	*Function*	Makes Bound-T use (also) the STABS symbol-table even if a DWARF symbol-table exists.
	Default	The STABS symbol-table is used only if there is no DWARF table.
-no_dwarf	*Function*	Denies the use of DWARF symbols. Bound-T will instead use STABS or ELF symbols if they exist.

Option		*Meaning and default value*
	Default	The DWARF symbol-table is used if it exists, in preference to STABS or ELF.
-no_elf_globals	*Function*	Denies the use of "global" ELF symbols when DWARF or STABS symbols exist.
	Default	The "global" ELF symbols are used to supplement DWARF and STABS symbols when available.

See also the option *-trace elf* in Table 18.

Instruction modelling options

The following table describes the options that control the modelling of the instructions in the target program to be analysed.

Table 13: Instruction Modelling Options for SPARC

Option		*Meaning and default value*
-abi	*Function*	Assume SPARC ABI rules for global registers (%g1 – %g4 are volatile across calls, %g5 – %g7 nonvolatile).
	Default	The default is *-no_abi* (see below).
-no_abi	*Function*	Assume all global registers are volatile across calls. This is the opposite of *-abi*.
	Default	This is the default.
-no_unsigned_cond	*Function*	Prevents the approximation of unsigned branch conditions (in the **Bicc** and **Ticc** instructions) by the corresponding signed branch conditions. Instead, the unsigned conditions are considered opaque. See section 3.8. This option makes the arithmetic analysis safer for combinations of signed and unsigned variables and computations, but the analysis becomes weaker and less able to find loop bounds.
	Default	The unsigned conditions are approximated by the corresponding signed conditions.
-par=const	*Function*	Asserts that no callee subprogram modifies the parameters that are passed in the stack, within the caller's frame.
	Default	The default is the opposite, *-par=var*, which see.
-par=var	*Function*	Asserts that a callee subprogram can modify the parameters that are passed in the stack, within the caller's frame.
	Default	This is the default.
-sethi_signed	*Function*	The immediate operand in a **SETHI** instruction (or a combined **SETHI-OR** pair) is taken as a signed two's complement 32-bit number.
	Default	The operand is taken as an unsigned 32-bit number.

Option		*Meaning and default value*
-sll_max=X	*Function*	Sets the maximum shift count (number of bit positions), *X*, for which an **SLL** instruction is modelled as a multiplication by $2^{shift\ count}$. Larger shifts give an opaque result. Note that large values of *X* may make the arithmetic analysis fail, because of the risk of overflow, even if the actual values in an execution of the target program do not cause overflow.
	Default	*-sll_max=10* giving a maximum multiplier of $2^{10} = 1024$.
-trap_base=X	*Function*	Set the trap base address to hexadecimal value *X*. Note that the last 12 bits of the address are required to be zeros. Bound-T needs the trap base address to find the trap handlers invoked explicitly by the **Ticc** instruction or implicitly by register window overflows or underflows.
	Default	Depends on the selected SPARC device. **Note**: To override the default, the *-trap_base* option must come **after** the *-device* option on the command line.
-code_base=X	*Function*	Set the code base address to hexadecimal value *X* to help handle dynamic (register indirect) jumps. All code to be analysed must lie at addresses greater or equal to *X*. Bound-T uses the code base address to reduce the numeric range of address expressions, by analysing them as offsets to the code base. This helps to avoid numeric overflow problems in the analysis.
	Default	Depends on the selected SPARC device. **Note**: To override the default, the *-code_base* option must come **after** the *-device* option on the command line.
-via_positive	*Function*	When the code contains an indexed branch through a table of branch addresses, using the code idiom explained in section 3.11, this option asserts that both source registers that are used to index the table contain non-negative or unsigned values. Depending on the computation of the table index this option may be necessary to help Bound-T locate the start of the address table. However, if the assertion is false, Bound-T may wrongly omit some targets of the indexed branch from the analysis, so the computed WCET bound may not cover all executions.
	Default	Bound-T determines the location and length of the address table from its arithmetic analysis of the address used in the instruction that loads an address from the table, supported only by the applicable assertions in the assertion file, if any.

Register window options

The option *–rw* enables the analysis of register-window usage and the register-file overflow and underflow traps. Section 2.2 explains this analysis. The following table describes the options that control the analysis. These options have no effect if the analysis is disabled (by default or by the option *–no_rw*). See also the option *–trap_base* in Table 13.

Table 14: Register Window Analysis Options for SPARC

Option		*Meaning and default value*
-rw *-rw_analysis*	*Function*	Enables the register window trap analysis. RW trap times are then included in the WCETs from Bound-T. The long form of this option is deprecated. The other options in this table are relevant only if the analysis is enabled with this option.
	Default	There is no analysis of the register window traps. Trap handling time is not included in the WCET bounds. The other options in this table have no effect on the analysis.
-no_rw *-no_rw_analysis*	*Function*	Disables the register window trap analysis. No RW trap times are included in the WCETs from Bound-T. The long form of this option is deprecated.
	Default	This is the default.
-max_win=X	*Function*	Sets the maximum number of register windows available in the system to the value *X*. This value should exclude the window reserved for trap handling. Note that *-max_win=1* forces Bound-T to assume a window trap for every execution of the **SAVE** and **RESTORE** instructions.
	Default	7 windows (*-max_win=7*). This corresponds to 8 register windows in the full register file of which one is reserved for traps.
-returns_trap	*Function*	Makes Bound-T assume a register window underflow trap for every return in the program.
	Default	Underflow traps are predicted by a global analysis.
-rw_calls	*Function*	Creates additional output that lists the calls that are important for the register window analysis. Output lines with the key *RW_First_Call* show the "first" calls that can cause overflow traps. Output lines with the keyword *RW_Deep_Call* show the "deep" calls that can cause underflow traps on return from the caller. This option is implicitly set by the generic option *-trace additional* that is described in the Reference Manual [1].
	Default	The calls are not listed.
-rw_overflow=X	*Function*	Sets the worst-case execution time of the register window overflow trap caused by a **SAVE** instruction to *X* cycles.
	Default	Bound-T analyses the trap handler to find a WCET bound. You may need the *-trap_base* option.
-rw_underflow=X	*Function*	Sets the worst-case execution time of the register window underflow trap caused by a **RESTORE** instruction to *X* cycles.
	Default	Bound-T analyses the trap handler to find a WCET bound. You may need the *-trap_base* option.

Floating point options

The following table describes the options that control the analysis of the Floating-Point Unit (FPU) and its interaction (sequential or concurrent operation and synchronization) with the Integer Unit (IU). The options fall into three groups:

– options to specify whether the FPU is sequential or concurrent with the IU,

– options to specify the execution times to be assumed for FP operations and apply for both sequential or concurrent FPUs,

– options to disable or enable the analysis of the blocking between the IU and a concurrent FPU. Section 2.3 explains the analysis.

If the analysis of IU-FPU blocking is disabled for a concurrent FPU then the WCET bounds will include FPU execution times only for floating-point comparison instructions such as **FCMPS**. As explained in section 2.3, floating-point comparisons are always executed synchronously with the IU. Thus their execution time is included in the WCET bound even for an FPU that runs all other operations concurrently and even if the blocking analysis is disabled.

Table 15: Floating-Point Options for SPARC

Option		*Meaning and default value*
-fpu=sequential *-fpu=concurrent*	*Function*	Specifies whether the FPU operates sequentially while the IU waits, or concurrently with the IU while the IU executes more instructions.
	Default	Depends on the selected SPARC device. **Note**: To override the default, the *-fpu* option must come **after** the *-device* option on the command line.
-fpu_typical	*Function*	Makes Bound-T use the "typical" times for floating-point operations, instead of the worst-case values. Note that this option may cause Bound-T to give too optimistic WCET values. It is the user's responsibility to judge if Bound-T's results are still valid for WCET analysis when this option is used.
	Default	Bound-T uses the worst-case times of floating-point operations as perhaps modified by the option *-fpu_time*.
-fpu_time=filename	*Function*	Makes Bound-T read FP operation times from the file called *filename*. See section 4.1 for the format of this file.
	Default	No file of FP times is read and the default FP operation times depend on the selected SPARC device. However, there are no default times for the extended- or quadruple-precision operations. If the target program uses such operations their execution time must be defined with this option.
-fp *-fp_analysis*	*Function*	Enables the analysis of the concurrent operation and blocking of the FPU and IU. Relevant only when *-fpu=concurrent*; has no effect when *-fpu=sequential*. The long form of this option is deprecated. Under this option, the floating-point computation and blocking times are included in the WCETs from Bound-T.
	Default	This the default when *-fpu=concurrent*.
-no_fp *-no_fp_analysis*	*Function*	Disables the analysis of the concurrent operation and blocking of the FPU and IU. Relevant only when *-fpu=concurrent*; has no effect when *-fpu=sequential*. The long form of this option is deprecated. Under this option, no floating-point computation or blocking times are included in the WCETs from Bound-T, with the exception of the floating-point comparison instructions like **FCMPS** for which the execution time is included in the WCET bounds because the IU always waits for these instructions to complete.

Option		*Meaning and default value*
	Default	The analysis is enabled by default.

See also the option *-trace fpu_reg* in Table 18.

Memory timing options

The following table describes the options that control the analysis of memory access timing for instruction fetches and load and store instructions.

At present Bound-T does not model or analyse cache memories. Thus, if your target has caches, for a safe WCET analysis you must use these options to define memory access timing that corresponds to a cache miss for each access.

For options that define a number of memory wait cycles, please note that the number of wait *cycles* is not always the same as the number of wait *states*; see Table 17 below. Note also that the number of wait cycles defined by a command-line option applies to all subprograms that are analysed, but can be overridden by assertions for specific subprograms or even for specific loops within a subprogram.

Table 16: Memory Timing Options for SPARC

Option		*Meaning and default value*
-read_ws=X	*Function*	Sets the number *X* of memory wait cycles that will be assumed for all memory reads (excluding stack references, alternate memory references and code fetches) .
	Default	Zero wait cycles (*-read_ws=0*).
-write_ws=X	*Function*	Sets the number *X* of memory wait cycles that will be assumed for all memory writes (excluding stack references and alternate memory references).
	Default	Zero wait cycles (*-write_ws=0*).
-code_ws=X	*Function*	Sets the number *X* of memory wait cycles that will be assumed for all code fetches.
	Default	Zero wait cycles (*-code_ws=0*).
-stack_read_ws=X	*Function*	Sets the number of memory wait cycles that will be assumed for all stack reads to the value *X*.
	Default	Zero wait cycles (*-stack_read_ws=0*).
-stack_write_ws=X	*Function*	Sets the number *X* of memory wait cycles that will be assumed for all stack writes.
	Default	Zero wait cycles (*-stack_write_ws=0*).
-alt_read_ws=X	*Function*	Sets the number *X* of memory wait cycles that will be assumed for all alternate memory reads (load instructions with an ASI).
	Default	Zero wait cycles (*-alt_read_ws=0*).
-alt_write_ws=X	*Function*	Sets the number *X* of memory wait cycles that will be assumed for all alternate memory writes (store instructions with an ASI).
	Default	Zero wait cycles (*-alt_write_ws=0*).

Memory access times defined with options (or with assertions in section 5.6) are in units of system clock *cycles*, not in units of memory wait *states*. Some types of memory may require several wait cycles for each wait state, or may require at least some constant number of wait cycles. The user must calculate the memory wait cycles for different types of memory by the formulas in Table 17 below, where *w* is the number of wait states of the memory. Note that this table may be valid only for the ERC32 processor; other conversions may apply to other SPARC implementations.

Table 17: Conversion Between Memory Wait States and System Clocks

Memory type	*Wait cycles for w wait states*
RAM	*w*
Boot-PROM	4*w*
I/O area	1 + *w*
Exchange memory	1 + *w*

For example, if the program is executed from Boot-PROM with $w = 2$ wait states, the option *-code_ws=8* should be used.

SPARC-specific -trace items

The following table shows the SPARC-specific additional tracing output items that can be requested with the generic Bound-T option *-trace* as explained in the Reference Manual [1]. By default no such tracing is enabled.

Table 18: SPARC-Specific *-trace* Items

-trace item	*Traced information*
elf	The process of reading ELF data and loading the program to be analysed. Displays each ELF element on standard output as it is read. This may help to understand and correct problems with the ELF structure.
ipoints	Each RapiTime ipoint (instrumentation point) and its "name" (unique numeric identifier) when detected in the target program. See section 4.7.
fpu_reg	Extends the generic *-trace decode* option to show also the FPU source and destination registers for each decoded instruction. Has no effect in the absence of *-trace decode*.
stabs	The process of reading symbol tables (debugging information) in the STABS format. This may help to understand and correct STABS parsing problems.

4.3 HRT Skeleton Analysis

The HRT architecture and Bound-T

The principles of the "HRT" (Hard-Real-Time) software architecture are explained in the user manual for the HRT mode of Bound-T [13]. Briefly, an HRT program is a multi-threaded, real-time program that consists of a number of active and concurrent *threads* or tasks, and a number of passive, protected *objects*. The threads interact via the protected

objects. A given protected object can act as a resource that is accessed with mutual exclusion (a *resource object*) or as a means for one thread to trigger (activate) another (a *synchronization object*).

The real-time structure of an HRT program can be described by an *HRT Execution Skeleton File* (ESF) that defines the threads and protected objects, defines the way in which they interact, and gives the WCET of the relevant code. The ESF can be used to analyse the schedulability of the whole program, which gives a static verification that all deadlines will be met.

As explained in reference [13], Bound-T can generate the ESF when given the option *-hrt* and the name of a "TPO file" instead of the names of root subprograms. The TPO file is written by the user and lists the threads and protected objects of the target program. Bound-T analyses the program to find the WCET bounds and the interaction between threads and protected objects, and emits the result as an ESF.

Generic coding rules

To enable HRT analysis by Bound-T the program's threads must be coded so that for each thread there is one subprogram that represents one activation of the thread. This is because Bound-T cannot analyse the eternal loop that is often required as the outermost structure of a thread, so the body of the loop must be separated into a subprogram.

HRT analysis of ORK programs

For HRT analysis, Bound-T/SPARC can be applied to programs that use the ORK kernel. Other compilers and kernels have not yet been tested with Bound-T/SPARC.

The TPO file must list the threads and protected object operations using the identifiers (symbols) assigned to them by the compiler and linker. Chapter 5 explains the mapping of source-code identifiers to linker symbols in general. If the program is written in Ada using the Ada tasking features, which map naturally to the HRT elements, additional name mappings are needed.

The following rules have been found by empirical observation of ORK binaries:

- *Thread*: For an Ada task with the Ada identifier *Foo*, the corresponding part of the linker symbol is *fooTK* (convert Ada identifier to lower case and add *TK* in upper case). Thus, if this task is contained in the package *Pak* and itself contains a subprogram *Sub*, the whole symbol for *Pak.Foo.Sub* is *pak_ _fooTK_ _sub*.
- *Protected object*: For an Ada protected object with the Ada identifier *Obj*, the corresponding part of the linker symbol is *objPT* (convert Ada identifier to lower case and add *PT* in upper case).
- *Operation of resource object*: If *Foo* is the Ada identifier of an Ada protected subprogram, the corresponding part of the linker symbol is *fooP* (convert Ada identifier to lower case and add *P* in upper case). Thus, if the Ada identifier of the protected object is *Obj* and the object is declared in the package *Pak*, the whole symbol for *Pak.Obj.Foo* is *pak_ _objPT_ _fooP*.
- *Signalling operation of a synchronisation object*: Same rules as for an operation of a resource object.
- *Waiting operation of a synchronisation object*: This corresponds to an Ada protected entry of an Ada protected object, and is a special case discussed below.

It seems that each protected subprogram *Obj.Foo* also has another symbol, *objPT_ _fooN*, that identifies the variant of *Obj.Foo* that is used when *Obj.Foo* is called from another operation of the same object and no lock is required.

Analysing waiting operations

In Ada, the waiting operation of an HRT synchronisation object is normally implemented as a protected entry of the Ada protected object, with a Boolean variable as the barrier condition. The task (thread) that needs to wait on the object would normally just call this entry. Unfortunately, it seems that the GNAT/ORK system does not name the entry subprogram in an easily understood way, and moreover implements such an entry-call by a GNAT run-time-system operation called *Protected_Single_Entry_Call* which gets the address of the entry subprogram as a parameter (a function pointer). Since Bound-T cannot analyse calls through function pointers, it cannot analyse such an entry-call. Instead, the target program must be coded and analysed in the following cumbersome way:

- Isolate the actions of the protected entry into a normal subprogram *Actions*, so that the entry itself is just of the following form:

```
protected body Obj is
   entry Wait when Barrier is
   begin
      Actions;
   end Wait;
   ...
end Obj;
```

- Isolate the entry call into a normal subprogram *Await* in the form:

```
procedure Await is
begin
   Obj.Wait;
end Await;
```

- Use Bound-T to compute the WCET bound for *Actions* in normal (non-HRT) mode.
- In the TPO file, name *Await* as the entry of the protected object, and also assert its WCET to be the WCET of *Actions*, plus the time required for the GNAT/ORK operation *Protected_Single_Entry_Call*, which must be measured or estimated in some way.

This work-around is not perfect if the entry calls other protected operations. The execution skeleton in the ESF will not show these calls, since Bound-T does not analyse the *Actions* subprogram as an HRT operation.

Handling the GNAT/ORK run-time system

The structure of the GNAT/ORK run-time system is relatively complex. When Bound-T is asked to analyse Ada code that performs run-time system calls, a number of run-time system subprograms must be excluded from the analysis by asserting their execution times. This issue is still under study, and we will give additional information in a later issue of this document or in a separate Application Note.

4.4 Output

This section describes the output from a Bound-T analysis of a SPARC program. It focuses on the SPARC-specific aspects; please refer to the Bound-T Reference Manual [1] for a generic description of the outputs.

Basic output format

Most Bound-T outputs, including warning and error messages, follow a common, basic format that contains the source-file name and source-line number that are related to the message. These output lines are explained in the Bound-T Reference Manual [1].

However, some compilers may not provide all the debugging information, depending on the optimization and debugging options. With such target programs, the Bound-T output will also be reduced, for example source-line numbers may be missing.

Units of measurement

Execution times (WCET bounds) are given in processor clock cycles.

Stack usage bounds are given in octets.

Outputs specific to the SPARC

Bound-T for the SPARC emits additional output lines explained in the following table. These lines report details of the SPARC-specific analyses: the analysis of pipeline blocks (stalls), IU/FPU blocks and register window traps. These results are included in the general WCET bounds so these additional output lines can usually be ignored.

As explained in the Reference Manual [1], in each output line a keyword in field 1 identifies the kind of output, fields 2 through 5 identify the program element, and the later fields contain the actual output. The table below is ordered alphabetically by the keyword column.

Table 19: Outputs for SPARC

Keyword (field 1)	*Explanation of fields 6 -*
Block_Wcet	*num* : *min* .. *max* : *local* : *callees* : *total* Reports the part of the WCET bound that comes from pipeline blocks (stalls, resource dependencies). There are two kinds of blocks in the SPARC: Integer Unit blocks and Floating-Point Unit blocks. IU blocks occur when an IU instruction uses a source operand register that is the destination register of the dynamically preceding instruction; this creates a one-cycle block. FPU blocks occur for concurrently executing FPUs for FPU load/store instructions and when an FPU instruction is still executing when the next FPU instruction is about to start execution, as explained in section 2.3. Both kinds of blocking are reported together in this output line. Sequentially executing FPUs do not cause blocking. The FPU execution time is included in the time the instruction spends in the IU. This output line reports the blocking that occurs (or could occur) in the execution path that defines the WCET bound for the current subprogram identified in fields 2 through 5. However, other execution paths (with a smaller or equal total execution time) may have more blocking. The fields have the following meaning:

Keyword (field 1)	*Explanation of fields 6 -*
	num is the number of blocked instruction pairs in this execution path. Each pair is counted once although it may be executed many times (in loops). *min* .. *max* is the range of blocking times per execution of a blocking instruction pair. If only IU blocks occur, the range is 1 .. 1; if FPU blocks occur, the range can be wider. *local* is the part of the WCET bound for the current subprogram that comes from blocking in this subprogram, excluding blocking in callees. *callees* is the part of the WCET bound for the current subprogram that comes from blocking in callee subprograms. *total* is *local* + *callees*, the total contribution of blocking to the WCET bound for the current subprogram.
RWin	*rwu_max* : *wd_min* .. *wd_max* : *overflows* : *underflows* : *time* Reports the results of the register window trap analysis (see section 2.2) for the current subprogram identified in fields 2 through 5. In this output line: *rwu_max* is the computed upper bound on the number of register windows in use at the start of the current subprogram. This includes the possible **SAVE** instruction that creates the stack frame for this subprogram, whether it comes immediately after the **CALL** in the calling subprogram or at the start of the current subprogram. *wd_min* .. *wd_max* are the bounds on the number of register windows pushed and popped by an execution of the current subprogram, including its callees. This number is called *win_depth* in section 2.2. *overflows* is the number of register window overflows included in the WCET bound for the current subprogram. It does not include overflows in callees. It can be zero or one, but nor more than one. *underflows* is the number of register window underflows included in the WCET bound for the current subprogram. It does not include underflows in callees. It can be zero or one, but not more than one. *time* is the part of the WCET bound for the current subprogram that comes from the register window overflow and underflow traps in the current subprogram. It does not include overflows or underflows in callees.
RW_Deep_Call	*wd_min* .. *wd_max* Emitted only when one or both of the options *-rw_calls* or *-trace additional* is used. Fields 2 through 5 identify a call; this output line reports that this call may push and pop so many register windows that the caller may be left with only its own register window in the register file, which means that a register window underflow trap occurs on return from the caller. See section 2.2. *wd_min* .. *wd_max* are the bounds on the number of register windows pushed and popped by the callee and deeper callees if any.
RW_Depth	*rwu_max* : *wd_min* .. *wd_max* Emitted only when one or both of the options *-rw_calls* or *-trace additional* is used. Reports intermediate results of the register window trap analysis. See the explanation of the *RWin* output line, above.

Keyword (field 1)	*Explanation of fields 6 -*
RW_First_Call	*wd_min .. wd_max* Emitted only when one or both of the options *-rw_calls* or *-trace additional* is used. Fields 2 through 5 identify a call; this output line reports that this call may be the first call that the caller executes and thus this call may cause a register overflow trap if the caller's *rwu_max* is large enough. See section 2.2. *wd_min .. wd_max* are the bounds on the number of register windows pushed and popped by the callee and deeper callees if any. However, they are not relevant for the occurrence of register window overflow traps at this point.
Trap_Handler	*trap description* Reports that the subprogram identified in fields 2 through 5 is considered the trap handler for the trap described in field 6. Currently Bound-T considers two kinds of traps: "Register Window overflow" and "Register Window underflow". See section 2.2 and the option *-trap_base* in section 4.2.

4.5 Warning Messages

The following table lists the Bound-T warning messages that are specific to the SPARC or that have a specific interpretation for this processor. The messages are listed in alphabetical order, perhaps slightly altered by variable fields in the message; such fields are indicated by *italic* text. The Bound-T Reference Manual [1] explains the generic warning messages, all of which may appear also when the SPARC is the target. The HRT-mode manual [13] explains the warnings that are specific to an HRT analysis. Section 4.7 explains the warning messages that may arise while exporting the target program for RapiTime analysis.

The specific warning messages refer mainly to unsupported or approximated features of the SPARC.

Table 20: Warning Messages

Warning Message		*Meaning and Remedy*
Alternate leaf return from trap	*Reasons*	The current subprogram ends with the normal return sequence for a leaf subprogram but the subprogram itself seems to be a trap handler. This is a contradiction.
	Action	Note that the analysis of this subprogram may be incorrect.
A privileged instruction in a normal subprogram	*Reasons*	The current subprogram contains a privileged instruction, but appears to be a normal subprogram and not a trap or interrupt handler.
	Action	If the application runs normal subprograms in “user” mode (not privileged), a trap will occur at the privileged instruction. The time and space bounds for the subprogram do not include this trap.

Warning Message		*Meaning and Remedy*
A SAVE instruction in a caller-saves subprogram	*Reasons*	The current **SAVE** instruction is out if place because it occurs in a caller-save subprogram, that is, the calling sequence already executes a **SAVE** for this subprogram.
	Action	Modify the program to avoid such code. If the current subprogram is a root subprogram that has been asserted to have the “caller-saves” property, remove this assertion.
Asserted RWU_Max (*A*) is less than computed RWU_Max (*C*)	*Reasons*	The assertion file asserts the property “rwu_max” to the value *A*, but Bound-T computes the maximum initial register-window usage of this subprogram to be a greater value *C*. The asserted value *A* is used in the analysis.
	Action	Check that the assertion is valid.
Assuming that Register Window traps may occur.	*Reasons*	See the warning “Not known how subprogram uses Register Windows”.
	Action	Ditto.
Callee parameter *P* maps to callee parameter *Q*	*Reasons*	While analysing the parameters that are passed in a given call, and in particular the stack location that the callee subprogram sees as a parameter *P* passed from the caller, Bound-T finds that the caller also sees this same stack location as a parameter *Q* that the caller's caller passes to the caller. This is unusual because when the callee uses *P* it is referring to a stack frame that is at least two levels away in the call-path. The *P* and *Q* symbols have the form “par*N*” where *N* is the octet offset from the stack pointer on entry to the subprogram (caller or callee). Possible causes for such code include tail calls in assembly language or nested subprograms.
	Action	You may want to check that Bound-T has analysed the stack-pointer and frame-pointer operations correctly for this particular call. The option *-trace param* may help.
Dynamically computed trap number *T* truncated to *N*	*Reasons*	The program contains a **Ticc** instruction where the trap number is a register operand (dynamically computed). Bound-T analysed the computation and found a single value *T* for the operand, however this value is outside the range for trap numbers and so Bound-T truncates the value to the seven least significant bits as defined in the SPARC architecture, giving trap number *N*.
	Action	Check the program to verify that the **Ticc** instruction works correctly and that the value *T* is correct. To avoid this warning, modify the program to make *T* a valid trap numbers without truncation.

Warning Message		*Meaning and Remedy*
Frame height becomes unknown	*Reasons*	The program contains an instruction that modifies the frame pointer register (**fp** = **r30**) in a way that Bound-T cannot analyse.
	Action	The analysis of computations involving variables accessed relative to the frame pointer may be unreliable. See Table 5 and the surrounding section.
Indirect jump to *A*	*Reasons*	The program contains a **JMPL** instruction that takes the target address from a register. Bound-T's analysis of the computation indicates that the register has the value *A*, which is thus the target address of the jump. The warning is issued because this analysis may be wrong in some cases.
	Action	Check that this **JMPL** instruction really has a single possible target.
Jump-and-link with destination register *R* is not seen as a call	*Reasons*	The current subprogram contains a **JMPL** instruction with destination register *R* which is neither **r0** nor **r15**. This means that the instruction saves a "return" address but not in the standard place (**r15**). Bound-T models the instruction as a jump and not as a call.
	Action	Check if Bound-T's model is correct.
Large literal *U* used signed = *S*	*Reasons*	The program contains a **SETHI-OR** instruction pair that loads a literal operand into a register, and this operand has the sign bit set, so that Bound-T uses the signed interpretation *S* instead of the unsigned interpretation *U*. The signed interpretation is considered only when the option *-sethi_signed* is in effect. This warning can be disabled with the option *-warn no_sign*.
	Action	Check that the signed interpretation is correct.
Leaf subprogram contains a call	*Reasons*	Based on its structure and calling sequence, the present subprogram seems to be a leaf subprogram (that is, it does not have a register window of its own) but even so it calls another subprogram. A leaf subprogram should not call other subprograms.
	Action	Check the program design and coding on this point.
Load for jump via table uses Immediate operand	*Reasons*	The program contains a **LD** instruction that loads a register from memory, followed by a **JMPL** to the address in the register. Bound-T interprets this as a switch-case structure that uses a table of addresses; see section 3.11. However, the **LD** instruction has an Immediate operand, which is unusual for this code idiom.
	Action	Check that Bound-T's interpretation of this code is correct.

Warning Message		*Meaning and Remedy*
Negative immediate address considered unknown	*Reasons*	The program contains an instruction that uses an immediate (literal) memory address that has the sign bit on, so that the value appears negative. The actual memory location that is accessed then depends on the actual memory size, not known to Bound-T, and so Bound-T considers the address unknown.
	Action	None. In theory, the situation may weaken Bound-T's analysis of loop bounds, but memory locations in this area are unlikely to contain loop counters.
No ELF symbol table found	*Reasons*	The option *-elf_symbols* told Bound-T to use the ELF symbol table, but the executable file does not contain an ELF symbol table.
	Action	Either get an executable file that does contain an ELF symbol table, or do not use the option *-elf_symbols*. For the GNU compilers, use the compiler option *-g*.
Normal leaf return from trap	*Reasons*	The current subprogram seems to be a trap handler but ends with the normal return sequence for a leaf subprogram. This is a contradiction.
	Action	Note that the analysis of this subprogram may be incorrect.
Normal return from trap	*Reasons*	The current subprogram seems to be a trap handler but ends with the normal return sequence for a non-leaf subprogram. This is a contradiction.
	Action	Note that the analysis of this subprogram may be incorrect.
No symbol tables found in the program	*Reasons*	The program under analysis does not contain any symbol table (debugging information) that Bound-T can use.
	Action	Obtain an ELF file with debugging symbols (for the GNU compilers, use the compiler option *-g*). Remove command-line options that deny the use of certain symbol tables. See Table 12.
Not known how subprogram uses Register Windows. Assuming that Register Window traps may occur.	*Reasons*	Bound-T is not sure if the present subprogram uses a register window of its own, but assumes that register window traps may occur. (Two warnings are emitted as shown at left.)
	Action	Check the code for the subprogram and possibly change it to use a standard call/return protocol.
No time analysis, so IU/FPU blocking ignored	*Reasons*	The option *-no_time* has disabled execution-time analysis, thus Bound-T decides not to perform the (unnecessary) concurrent-FPU timing analysis although it has not been disabled with *-no_fp*.
	Action	To suppress this message use also the option *-no_fp* whenever you use *-no_time*.

Warning Message		*Meaning and Remedy*
No time analysis, so register window traps ignored	*Reasons*	The option *-no_time* has disabled execution-time analysis, thus Bound-T decides not to perform the (unnecessary) register-window analysis although it was enabled with *-rw*.
	Action	To suppress this message do not use the option *-rw* together with *-no_time*.
Not sure if a register window is used	*Reasons*	Bound-T is not sure if the present subprogram uses a register window of its own, but assumes it does not.
	Action	None. The "window depth" (*win_depth* in section 2.2) may be underestimated by one.
Not sure if *S* is a leaf subprogarm.	*Reasons*	Bound-T is not sure if subprogram S is a leaf subprogram, that is, whether or not it uses its own register window. Bound-T assumes that *S* is a leaf subprogram and uses the same register window as its caller.
	Action	None. The situation probably has no effect on the analysis. It may mean that a later warning "Leaf subprogram contains a call" for subprogram *S* is spurious. It may also mean that a call to subprogram *S* is wrongly considered not to cause a register window overflow.
Not sure if *S* is a trap handler.	*Reasons*	Bound-T is not sure if subprogram S is a trap handler subprogram, but assumes that it is.
	Action	None. The situation probably has no effect on the analysis. It may mean that a call to subprogram *S* is wrongly considered not to cause a register window overflow.
Object file *problem*	*Reasons*	There is some *problem* in the executable file named on the command line; the file is perhaps not in a format that Bound-T supports.
	Action	Get an executable file that Bound-T can analyse.
Property *P* has no valid upper bound. Using zero.	*Reasons*	An assertion constrains the value of property *P* (*eg.* the "code_ws" property) but does not place an upper bound on the value, which means that there is no upper bound on the execution time. Bound-T uses a zero value for this property.
	Action	Correct the assertion file.
Register Window usage unclear	*Reasons*	Bound-T cannot classify the current subprogram as a leaf, self-save, caller-save or trap handler sub-program. These terms are defined in section 3.11.
	Action	Note that the analysis of register-window traps for this subprogram and for calls to this subprogram may be inaccurate.

Warning Message		*Meaning and Remedy*
Resolving jumps via constant address table at *A* .. *B*	*Reasons*	The program contains the code idiom (instruction pattern) that implements a switch/case statement with a table of addresses; see section 3.11. Bound-T's analysis of the index computation indicates that the address table occupies the addresses *A* .. *B*, but this analysis may be wrong in some cases. Moreover, Bound-T assumes that the address table is constant and not modified during program execution.
	Action	Check that the addresses *A* .. *B* indeed contain the whole address table for a switch/case and that this address table is constant.
Return from interrupt from Trap call	*Reasons*	The current subprogram ends with the return-from-interrupt sequence but the subprogram itself seems to be a trap handler. This is contradictory.
	Action	Note that the analysis of this subprogram may be incorrect.
Return from trap from Interrupt call	*Reasons*	The current subprogram ends with the return sequence for a trap handler but the subprogram itself seems to be an interrupt handler. This is a contradiction.
	Action	Note that the analysis of this subprogram may be incorrect.
RW Overflow (Underflow) trap became bounded in FP analysis	*Reasons*	For a concurrent FPU, before the FPU timing analysis Bound-T was unable to bound the WCET of the indicated trap handler, but after the FPU analysis a WCET bound was found. This is surprising.
	Action	Please report this event to Tidorum Ltd.
RW Overflow (Underflow) trap became unbounded in FP analysis	*Reasons*	For a concurrent FPU, before the FPU timing analysis Bound-T was abled to bound the WCET of the indicated trap handler, but after the FPU analysis a WCET bound was not found. This is surprising.
	Action	Please report this event to Tidorum Ltd.
RW Overflow (Underflow) trap time-bound changed in FP analysis	*Reasons*	For a concurrent FPU, the analysis of the FPU timing (section 2.3) changed (increased) the WCET bound for the indicated trap handlers (because the handler contains a significant amount of such blocking).
	Action	The amount of FPU blocking reported for subprograms that can cause register window traps may be overestimated, because the FPU block analysis was based on the initial (smaller) WCET bounds for these trap handlers.
SAVE after RESTORE is not modelled	*Reasons*	The current instruction is a **SAVE** but the current subprogram has already executed a **RESTORE**. Bound-T cannot model this manipulation of the register windows.
	Action	Modify the program to avoid such code.

Warning Message		*Meaning and Remedy*
STABS N_Sline record with no base address, ignored	*Reasons*	The executable ELF file contains a STABS source-line record (N_Sline record) that is not in the context of a subprogram and so has no base address, therefore Bound-T cannot use the record. Probably the file is in some variant of ELF/STABS that Bound-T does not understand.
	Action	Try to get an executable file in the ELF format that Bound-T supports, preferable with DWARF debugging information.
STABS register number *N* is not modelled	*Reasons*	The executable ELF file contains a STABS symbol record describing a variable as located in register number *N*, in the range 1 .. 31, but Bound-T does not model SPARC register number *N* as a cell in the current register-window context.
	Action	None, but the variable mapped to register *N* may not be usable in assertions.
Stack frame location and size become unknown	*Reasons*	The current **SAVE** instruction occurs in a context where Bound-T is unable to model the changes in the register windowing, or specifies a destination register other than **sp** = **r14**.
	Action	Modify the program to avoid such code.
Stack height becomes unknown	*Reasons*	The program contains an instruction that modifies the stack pointer register (**sp** = **r14**) in a way that Bound-T cannot analyse.
	Action	The result of the stack usage analysis should not be considered reliable. See section 2.5.
Time asserted. Assuming zero blocking time.	*Reasons*	The WCET bound for the current subprogram is asserted in the assertion file, not computed; therefore, the pipeline blocking in the subprogram was not analysed, and no blocking is assumed.
	Action	Note that the blocking time reported for this subprogram (*Block_Wcet* output lines) does not correspond to the actual code of the subprogram.
Time is asserted but not window-depth; using zero	*Reasons*	The assertion file asserts an execution time for this subprogram, but does not assert any value for the “win_depth” property, so Bound-T uses zero as the value of “win_depth”.
	Action	Correct the assertion file.
Too many RESTORE levels	*Reasons*	The current **RESTORE** instruction does not match an earlier **SAVE** instruction in the current subprogram or in the calling sequence for the current subprogram. Bound-T cannot model the effect of this **RESTORE** instruction.
	Action	Modify the program to avoid such code.
Too many SAVE levels	*Reasons*	The current subprogram (or its calling sequence) has already executed a **SAVE** instruction. Bound-T cannot model the effect of the current second (or third, etc.) **SAVE** instruction.
	Action	Modify the program to avoid such code.

Warning Message		*Meaning and Remedy*
Trap 0 taken as return	*Reasons*	The program contains a **Ticc** instruction for trap number zero, which usually means a software reset and reboot. Bound-T models this instruction as a return from the current subprogram.
	Action	Note that the time and space bounds for the current subprogram do not include the trap handling.
UNIMP instruction taken as return	*Reasons*	The program contains an **UNIMP** instruction that is not within an alternate call sequence. Bound-T models this instruction as a return from the current subprogram.
	Action	Note that the time and space bounds for the current subprogram do not include the trap that results from executing **UNIMP**.
Zero weight for FPU delay on step-edge *E*	*Reasons*	For a concurrent FPU, the heuristic formula that Bound-T uses to assign weights to flow-graph edges that may account for FPU blocking delays has assigned zero weight to edge number *E*. This may degrade the assignment of FPU blocking delays to edges and so lead to a pessimistic WCET bound.
	Action	None.

4.6 Error Messages

The following table lists the Bound-T error messages that are specific to the SPARC or that have a specific interpretation for this processor. The messages are listed in alphabetical order, perhaps slightly altered by variable fields in the message; such fields are indicated by *italic* text. The Bound-T Reference Manual [1] explains the generic error messages, all of which may appear also when the SPARC is the target. The HRT-mode manual [13] explains the error messages that are specific to an HRT analysis. Section 4.7 explains the warning messages that may arise while exporting the target program for RapiTime analysis.

Table 21: Error Messages

Error Message		*Meaning and Remedy*
Alternate leaf return from *C* call	*Problem*	The current subprogram ends with the alternate return sequence for a leaf subprogram but the subprogram itself seems to be of type *C*: a subprogram to be called with the normal sequence or a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.

Error Message		*Meaning and Remedy*
Alternate return from *C* call	*Problem*	The current subprogram ends with the alternate return sequence for a non-leaf subprogram but the subprogram itself seems to be of type *C*: a subprogram to be called with the normal sequence or a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
Alternate leaf return in the callee's window	*Problem*	The current subprogram ends with the alternate return sequence for a leaf subprogram, but the subprogram seems to be using its own register window at this point. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the register window management.
	Solution	Modify the program to follow the calling conventions.
Alternate leaf return in a trap window	*Problem*	The current subprogram ends with the alternate return sequence for a leaf subprogram, but the subprogram seems to be using a trap-handler register-window at this point. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the register window management.
	Solution	Modify the program to follow the calling conventions.
Asserted negative minimum (maximum) window depth, using zero.	*Problem*	The assertion file asserts the property *win_depth* for a subprogram, but gives a negative minimum (maximum) bound for the value of this property.
	Reasons	An error in the assertion file.
	Solution	Correct the assertion file.
Asserted RWU_Max is out of range: *bounds on RW*	*Problem*	The asserted property *rwu_max* for a subprogram is either greater than the maximum window usage of the system, or lower than or equal to zero.
	Reasons	An error in the assertion file.
	Solution	Correct the assertion file.
Asserted too large minimum (maximum) window depth, using *D*.	*Problem*	The assertion file asserts the property *win_depth* for a subprogram, but gives a too large minimum (maximum) bound for the value of this property. A bound is too large if it exceeds the total number *D* of available register windows (option *-max_win*).
	Reasons	An error in the assertion file or in the command line.
	Solution	Correct the assertion file or the command-line option *-max_win*.
Call instruction at return	*Problem*	The last instruction in the subprogram, in the return sequence itself, is a **CALL** instruction, which violates the calling protocols.
	Reasons	Error in the program being analysed.
	Solution	Correct the return sequence of the subprogram; move the **CALL** to occur before the return sequence.

Error Message		*Meaning and Remedy*
Cannot open FPU-time file for reading	*Problem*	The file with FP operation times, named in the option *-fpu_time*, could not be opened for reading.
	Reasons	The file-name may be wrong (such a file does not exist) or the file permissions may not allow reading.
	Solution	Check and correct the file-name or the file permissions.
Cannot read file	*Problem*	Bound-T could not read the executable file (target program) named on the command line.
	Reasons	The file permissions do not let the user read the file, or the file type does not allow the access method that Bound-T uses.
	Solution	Correct the type or permissions of the file.
Does not seem to be a SPARC/ELF file	*Problem*	The given executable file (target program) does not seem to be an ELF file for a SPARC program.
	Reasons	The executable file uses some other format, or does not contain a SPARC program, or uses an ELF variant that Bound-T does not support.
	Solution	Check the compiler and linker options. If they are correct, contact Tidorum Ltd.
Dynamic call at return	*Problem*	The last instruction in the subprogram, in the return sequence itself, is a **JMPL** instruction that implements a dynamic call. This violates the calling protocols.
	Reasons	Error in the program being analysed.
	Solution	Correct the return sequence of the subprogram; move the **JMPL** to occur before the return sequence.
Dynamic jump or call resolved to invalid address *base* + *offset*	*Problem*	The program contains a **JMPL** instruction with a computed target address. Bound-T's analysis of the computation of the target address gives a result that exceeds the range of SPARC code addresses. The result is expressed as the sum of a constant *base* (displayed in hexadecimal) and a computed *offset* (displayed in decimal).
	Reasons	The analysis of the target address computation is probably wrong, possibly because of undetected aliasing (pointers), or because the computation depends on 32-bit overflow (address wrap-around) that Bound-T does not model.
	Solution	Replace the dynamic jump or call by a static jump or call.
File does not exist	*Problem*	The executable file for the target program, named on the command line, was not found.
	Reasons	The file-name is wrong, or some directory in the path to the file does not allow access.
	Solution	Correct the file-name or the permissions on the path.
FP instruction not recognised : *M*	*Problem*	The file with FP operation times, named in the option *-fpu_time*, contains an unrecognised operation mnemonic *M*.
	Reasons	Error in the file.
	Solution	Correct the file.

Error Message		*Meaning and Remedy*
FPU-time text not understood: *text*	*Problem*	The file with FP operation times, named in the option *-fpu_time*, contains a line with some kind of syntax error. The line contains the given *text*.
	Reasons	Error in the file.
	Solution	Correct the file.
Ignoring asserted “caller_save” values (must be single non-negative value)	*Problem*	An assertion gives an invalid value or range of values to the *caller_save* property. This property must be given a single non-negative value.
	Reasons	Error in the assertion file.
	Solution	Correct the assertion file.
Ignoring asserted “call” values (must be single value *A* .. *B*)	*Problem*	An assertion gives an invalid value or range of values to the *call* property. This property must be given a single value in the range *A* .. *B*.
	Reasons	Error in the assertion file.
	Solution	Correct the assertion file.
Instruction address *A* exceeds segment boundaries	*Problem*	The program seems to reach an instruction at an address *A* that is not in any code segment, that is, it is not present in the memory image at load time. The analysis stops at this point.
	Reasons	The most common reason is that the actual Trap Base Address in the target program differs from the address that Bound-T uses, either by default from the chosen SPARC device or as given in the *-trap_base option*. This makes Bound-T go to the wrong address to find the code for a trap handler. Another possible reason is that Bound-T is exploring an execution path that is impossible or mistaken, perhaps because a switch/case address table is overestimated. The analysis of some switch/case statements needs the *-via_positive* option. It can also happen that the execution path being explored is possible but the program will place some instruction at address *A* before this address is reached during execution.
	Solution	Check your *-device* and *-trap_base* options. Note that *-trap_base* must come **after** *-device*. Use the option *-via_positive* to help the analysis of switch/case statements. Ensure that all code is statically present in the program's load image, not created or moved during execution.

Error Message		*Meaning and Remedy*
Invalid instruction *or* Invalid instruction taken as return	*Problem*	The program seems to reach an illegal instruction; an instruction word that does not encode a valid instruction in the chosen version of the SPARC architecture.
	Reasons	There are many possible reasons. There may simply be an error in the program. Perhaps the program is meant for a later version of the SPARC architecture and so contains instructions that have been added after SPARC V7, V8 or V8E (depending on the *-device* chosen). Perhaps Bound-T is exploring an execution path that is impossible or mistaken, maybe because a switch/case address table is overestimated. Perhaps the path is possible but the program will place some valid instruction here, before execution reaches this address.
	Solution	Check that your SPARC program is compiled for the chosen *-device*. Check that Bound-T has analysed dynamic jumps correctly. Ensure that all code is statically present in the program's load image, not created or moved during execution.
JMPL is last instruction in subprogram	*Problem*	The last instruction in the subprogram, in the return sequence itself, is a **JMPL** instruction, which violates the SPARC coding rules.
	Reasons	Error in the program being analysed.
	Solution	Correct the code of the subprogram.
Jump via table finds invalid address *A*	*Problem*	The program seems to contain a switch/case construct that is encoded with a table of addresses as explained in section 3.11 but the table contains a value *A* that is not a valid code address.
	Reasons	The decoding is probably wrong; perhaps Bound-T has overestimated the size of the table so that the value *A* is not taken from the table, but from some point before or after the table.
	Solution	Simplify the switch/case construct or replace it with conditional statements (if-then-elsif ...).
Jump via table slot at address *A* that exceeds segment boundaries	*Problem*	The program seems to contain a switch/case construct that is encoded with a table of addresses as explained in section 3.11 but the table extends to an address *A* that is not in any code segment, that is, it is not present in the memory image at load time.
	Reasons	The decoding is probably wrong; perhaps Bound-T has overestimated the size of the table. Alternatively, the table is not constant (statically loaded) but is filled dynamically by the program before the switch/case statement is executed.
	Solution	Simplify the switch/case construct or replace it with conditional statements (if-then-elsif ...). Ensure that all address tables are statically present in the program's load image, not created during execution.

Error Message		*Meaning and Remedy*
Mismatch of call kind : New *N* : Current *C*	*Problem*	A subprogram is called using different calling protocols (normal, alternate, trap or interrupt; see section 3.11). Before this point in the analysis, the calling protocol appeared to be protocol *C*; now, the program indicates that the protocol is *N*.
	Reasons	An error in the program, or a subprogram that somehow adapts to different calling protocols.
	Solution	Change the program to use one calling protocol for each subprogram.
No -device was specified	*Problem*	The command line contains no *-device=name* option to select the SPARC device.
	Reasons	Mistake on the command line.
	Solution	Add a *-device=name* option to the command line.
Mismatch of caller-push : New *N* : Current *C*	*Problem*	A caller-save subprogram is called with a delayed **SAVE** instruction that pushes a stack-frame of a different size (*N*) than the size (*C*) used in earlier calls of the same subprogram. *N* and *C* are expressed as bounds on the symbolic variable *size*.
	Reasons	Perhaps the subprogram is written to use a stack frame of variable size, defined by the caller.
	Solution	Change the program to make the called subprogram a self-save subprogram (with its own **SAVE**), or to push the same size of frame in all caller-save calls.
Mismatch of RW kind : New *N* : Current *C*	*Problem*	A subprogram is called using different register window protocols (caller-save, self-save, leaf or trap; see section 3.11). Before this point in the analysis, the RW protocol appeared to be protocol *C*; now, the program indicates that the protocol is *N*.
	Reasons	An error in the program, or a subprogram that somehow adapts to different register-window protocols.
	Solution	Change the program to use one register window protocol for each subprogram.
Normal leaf return from *C* call	*Problem*	The current subprogram ends with the normal return sequence for a leaf subprogram but the subprogram itself seems to be of type *C*: a subprogram called with the alternate sequence or a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
Normal return from *C* call	*Problem*	The current subprogram ends with the normal return sequence for a non-leaf subprogram but the subprogram itself seems to be of type *C*: a subprogram called with the alternate sequence or a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.

Error Message		*Meaning and Remedy*
Normal leaf return in the callee's window	*Problem*	The current subprogram ends with the normal return sequence for a leaf subprogram, but the subprogram seems to be using its own register window at this point. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the register window management.
	Solution	Modify the program to follow the calling conventions.
Normal leaf return in a trap window	*Problem*	The current subprogram ends with the normal return sequence for a leaf subprogram, but the subprogram seems to be using a trap-handler register-window at this point. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the register window management.
	Solution	Modify the program to follow the calling conventions.
Object file *problem*	*Problem*	There is some *problem* in the executable file named on the command line.
	Reasons	The file is perhaps not in a format that Bound-T supports.
	Solution	Get an executable file that Bound-T can analyse.
Only one token in this FPU-time line : *text*	*Problem*	The file with FP operation times, named in the option *-fpu_time*, contains a line that has only one textual token, whch is a syntax error. The line contains the given *text*.
	Reasons	Error in the file.
	Solution	Correct the file.
Patch address *A* exceeds segment boundaries	*Problem*	The patch file, named in the *-patch* option, specifies patching at address *A* but the address is not in any code segment, that is, it is not present in the memory image at load time.
	Reasons	Error in the patch file. Possibly the patch is meant for another executable, with different address ranges.
	Solution	Correct the patch file. Only addresses that are present in the loaded memory image can be patched.
Patch address *A* is not 32-bit aligned	*Problem*	The patch file, named in the *-patch* option, specifies patching at address *A* but the address is not aligned at a word boundary (not a multiple of 4 octets).
	Reasons	Error in the patch file.
	Solution	Correct the patch file. All patch addresses must be word aligned (multiples of 4).
Patch data invalid: *data*	*Problem*	The patch file, named in the *-patch* option, provides the given patch *data* but this could not be interpreted as a 32-bit hexadecimal word.
	Reasons	Error in the patch file.
	Solution	Correct the patch file. All patch data must be written in hexadecimal and fit in 32 bits (unsigned).
Patch data or params invalid	*Problem*	The patch file, named in the *-patch* option, contains a line that is in error.

Error Message		*Meaning and Remedy*
	Reasons	Error in the patch file.
	Solution	Correct the patch file. See section 4.1.
Return from interrupt from *C* call	*Problem*	The current subprogram ends with the return sequence for an interrupt handler but the subprogram itself seems to be of type *C*: not a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
Return from interrupt from *K* subprogram	*Problem*	The current subprogram ends with the return sequence for an interrupt handler but the subprogram itself seems to be of kind *C*: not a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
Return from interrupt from normal subprogram in *V*	*Problem*	The current subprogram ends with the return sequence for an interrupt handler but the subprogram seems to be using a normal register-window view *V* at this point. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the register window management.
	Solution	Modify the program to follow the calling conventions.
Return from trap from *C* call	*Problem*	The current subprogram ends with the return sequence for a trap handler but the subprogram itself seems to be of type *C*: not a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
Return from trap from *K* subprogram	*Problem*	The current subprogram ends with the return sequence for a trap handler but the subprogram itself seems to be of type *K*: not a trap or interrupt handler. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
Return from trap from normal subprogram in *V*	*Problem*	The current subprogram ends with the return sequence for a trap handler but the subprogram seems to be using a normal register-window view *V* at this point. This is a contradiction.
	Reasons	The program is written that way, or Bound-T is confused about the register window management.
	Solution	Modify the program to follow the calling conventions.
Return with restore from a *K* subprogram	*Problem*	The current subprogram ends with a return sequence that includes a **RESTORE** instruction, as proper for a self-save or caller-save subprogram, but the subprogram itself seems to be of a different kind *K*. This is a contradiction.

Error Message		*Meaning and Remedy*
	Reasons	The program is written that way, or Bound-T is confused about the type of the subprogram.
	Solution	Modify the program to follow the calling conventions.
RETT instruction out of context	*Problem*	There is a **RETT** instruction in the program without a preceding **JMPL** instruction or in some other context that does not match the standard return-from-trap instruction sequence.
	Reasons	An error in the program.
	Solution	Correct the program.
RWU_Max is asserted, but without upper bound: *bounds on RW*	*Problem*	The assertion file asserts the property *rwu_max* for a subprogram, but places no upper bound on the value, which is meaningless.
	Reasons	Error in the assertion file.
	Solution	Correct the assertion file.
SAVE for call does not set the stack pointer	*Problem*	This **SAVE** instruction is the delayed instruction for a subprogram call but the destination register is not the stack pointer **sp** = **r14**. This violates the calling conventions.
	Reasons	The program is written that way.
	Solution	Change the program to use the normal calling conventions.
SAVE for call increases the stack pointer	*Problem*	This **SAVE** instruction is the delayed instruction for a subprogram call but its effect is to increase, not decrease, the value of the stack pointer **sp** = **r14**. This violates the calling conventions.
	Reasons	The program is written that way.
	Solution	Change the program to use the normal calling conventions.
SAVE for call is not static	*Problem*	This **SAVE** instruction is the delayed instruction for a subprogram call but its effect on the stack pointer **sp** = **r14** is defined by a dynamic computation (register operand) instead of a static value (immediate operand). Bound-T cannot determine the size of the stack frame that is allocated for the callee.
	Reasons	The program is written that way, perhaps because the stack frame is too large to be encoded as an immediate operand.
	Solution	Change the program to avoid this kind of code.
Strange JMPL taken as return	*Problem*	The program contains a **JMPL** instruction that does not match any of the standard ways of using a **JMPL**: as a dynamic jump, a dynamic call, or a return from a subprogram, trap or interrupt. Bound-T assumes that the instruction implements a return from the current subprogram.
	Reasons	The target program is written that way.
	Solution	Change the program to use **JMPL** in ways that Bound-T can handle.

Error Message		*Meaning and Remedy*
The instruction word *W* at the normal return point *A* is not a valid SPARC instruction	*Problem*	The program contains a subprogram call that seems to return to an address *A* that contains the word *W* which is not a valid SPARC instruction nor the special "unimplemented" instruction **UNIMP**.
	Reasons	The call and the callee do not follow the standard calling protocol, or the program itself dynamically puts a valid instruction at address *A* before executing the call.
	Solution	Ensure that the standard calling protocol is followed and that all code is statically present in the program's load image, not created or moved during execution.
The normal return point *A* contains no instruction.	*Problem*	The program contains a subprogram call that seems to return to an address *A* that is not in the code segment of the program (not loaded with an instruction).
	Reasons	The call and the callee do not follow the standard calling protocol, or the program itself dynamically puts some instructions at address *A* before executing the call.
	Solution	Ensure that the standard calling protocol is followed and that all code is statically present in the program, not created or moved during execution.
Trap instruction in delay slot	*Problem*	The program contains a **Ticc** (trap on integer condition) instruction in an illegal context (in the delay slot of a control-transfer instruction).
	Reasons	The program is written like that.
	Solution	Correct the target program.
Unable to bound the WCET of the RW Overflow (Underflow) handler. Using WCET = 0.	*Problem*	Bound-T is unable to bound the WCET of a trap handler for the register-window trap analysis. This analysis is explained in section 2.2.
	Reasons	The trap handler is written in a way that Bound-T cannot analyse. There should be other error reports, before this one, that explain the problem in more detail.
	Solution	User the option *-rw_overflow* or *-rw_underflow* to set the WCET of the trap handler, or disable the register-window trap analysis entirely.
Unaligned DOUBLEWORD in IU register *R*	*Problem*	The program uses IU register *R* as a doubleword operand, but *R* is an odd number.
	Reasons	The program is written in that way.
	Solution	Correct the program to use even register numbers for doubleword data.
Unaligned *T* in FPU register *F*	*Problem*	The program uses FPU register *F* as an operand of the multi-word floating-point type *T*, but *F* is not a multiple of the number of 32-bit words in *T*.
	Reasons	The program is written in that way.
	Solution	Correct the program to use properly aligned register numbers for multi-word floating-point data (extended and quadruple-precision data).

Error Message		*Meaning and Remedy*
Unexpected end of ELF file	*Problem*	The executable target program file named on the command line ends unexpectedly, at a point where more data is expected.
	Reasons	The file may be damaged, or it may be in a format that Bound-T does not support.
	Solution	Correct the file.
Unknown device: *Arg*	*Problem*	The device-name given in the command-line argument *Arg* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line.
Unknown -fpu option: *Arg*	*Problem*	The value given for the *-fpu* option in the command-line argument *Arg* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line.
Unknown or invalid FP instruction	*Problem*	The current instruction seems to be a floating-point instruction that Bound-T does not support or for which Bound-T does not know the execution time.
	Reasons	The instruction specifies a combination of FP operand types that is not supported in the chosen SPARC architecture (V7, V8, V8E), or applies to extended-precision or quadruple-precision operands and the execution time of this instruction has not been defined with the *-fpu_time* option.
	Solution	Modify the program to use only supported FP instructions, or use the *-fpu_time* option to define the execution time.
Unknown -par option: *Arg*	*Problem*	The value given for the *-par* option in the command-line argument *Arg* is not recognised.
	Reasons	Error in the command line.
	Solution	Correct the command line.
Unknown register-view on entry, assuming caller-view	*Problem*	Bound-T is starting to analyse a subprogram but does not know if this is a self-save or a caller-save subprogram. It assumes the former (self-save) so that the subprogram begins its execution using the caller's register window.
	Reasons	This error should never occur.
	Solution	Please inform Tidorum Ltd.
Value *V* for *opt* is out of range; should be *A* .. *B*: *Arg*	*Problem*	The value *V* given in the command-line argument *Arg* for the option *opt* is not an acceptable value for this option. The acceptable values are between *A* and *B*, inclusive.
	Reasons	Error in the command line.
	Solution	Correct the command line.
Value for *opt* should be a natural number (base 10): *Arg*	*Problem*	The value given in the command-line argument *Arg* for the option *opt* is not an unsigned, decimal number or has too many digits.
	Reasons	Error in the command line.
	Solution	Correct the command line.

Error Message		*Meaning and Remedy*
Value is not a multiple of 0x1000: *Arg*	*Problem*	The value given in the command-line argument *Arg* is not acceptable because only multiples of 1000 (hex) are allowed.
	Reasons	Error in the command line.
	Solution	Correct the command line.
Value is out of range or not a hexadecimal number: *Arg*	*Problem*	The value given in the command-line argument *Arg* is not an unsigned, hexadecimal number or has too many hexadecimal digits.
	Reasons	Error in the command line.
	Solution	Correct the command line.

4.7 RapiTime Export

RapiTime

RapiTime from Rapita Systems Ltd [14] is a timing analysis tool that uses measurements (tests) to compute execution time distributions and to estimate execution-time bounds. RapiTime has different ways to measure execution times with high resolution of program parts, down to basic blocks. One way is to instrument the source code with instrumentation points or *ipoints*, typically calls to a specific subprogram that records the execution time at this point. The subprogram has a single parameter, an integer that uniquely identifies the ipoint and is recorded together with the time. The value of this integer is called the *name* of the ipoint.

After the test is executed, RapiTime analyses the trace (log) of ipoint records to find out the execution paths (the paths in the "ipoint graph") and the the execution time at each ipoint. For this analysis of the time measurements, RapiTime needs to know the program structure in terms of the machine-level control-flow graphs and call graphs and the location of the ipoints in these graphs. Bound-T for SPARC can optionally export the analysed subprograms as an XML file that defines this structure.

The structure of the XML file is defined in [15].

RapiTime options

The following command-line options control the RapiTime export function in Bound-T.

Table 22: RapiTime Export Options

Option		*Meaning and default value*
-rapitime=filename	*Function*	Exports the analysed program parts to a RapiTime XML file of the given *filename*. If a file with this name already exists, it is overwritten.
	Default	No export.
-ipoint=name or *-ipoint=address*	*Function*	Defines the subprogram of the given *name*, or the given hexadecimal entry *address*, as the RapiTime instrumentation routine. Every call of this subprogram is an ipoint.

Option		*Meaning and default value*
	Default	None. If a RapiTime file is exported either this option or *-no_ipoints* must be defined.
-no_ipoints	*Function*	Disables the detection and export of ipoints, as an alternative to the *-ipoint* option. However, the resulting file is not useful for a RapiTime analysis because it does not show the location of the instrumentation points.
	Default	None. If a RapiTime file is exported either this option or the *-ipoint* option must be defined.

See also the option *-trace ipoints* in Table 18.

Warning messages

While exporting a RapiTime file, Bound-T can emit some warnings as explained in the following table.

Table 23: RapiTime Export Warning Messages

Warning Message		*Meaning and Remedy*
No RapiTime <link> for unresolved dynamic call	*Reasons*	The program contains a dynamic call (or a trap instruction) that Bound-T could not resolve, so the possible callees are unknown and the call cannot be represented as a <link> element in the RapiTime output.
	Action	Change the program to uses static calls, or assert the possible callees.

Error messages

While exporting a RapiTime file, Bound-T can emit some error messages as explained in the following table.

Table 24: RapiTime Export Error Messages

Error Message		*Meaning and Remedy*
Cannot create RapiTime file named "*name*"	*Problem*	Bound-T could not create a file with the given *name* to hold the RapiTime XML form of the analysed subprograms.
	Reasons	File or directory permissions prevent the creation.
	Solution	Change the permissions to allow creation or change the *name* to create the file in a directory that allows it.
Ipoint subprogram not found	*Problem*	The subprogram named in the option *-ipoint=name* was not found in the symbol-table of the target program.
	Reasons	The name may be mistyped, or the compiler may have “mangled” the source-code name into a different linkage name.
	Solution	Check the symbol table and correct the name, or use the option form *-ipoint=address*.

Error Message		*Meaning and Remedy*
No -ipoint defined for RapiTime	*Problem*	The instrumentation routine is unknown.
	Reasons	The command line has neither an *-ipoint* option nor a *-no_ipoints* option.
	Solution	Correct the command line.
RapiTime ipoint-name unknown: *interval*	*Problem*	While analysing the “name” (parameter value) of this RapiTime ipoint call, Bound-T did not find a single value, but an *interval* of values, or no limits at all. The *interval* has the form *min* <= name <= *max*, where *min* and *max* are the computed lower and upper limits, or absent if no limit in that direction was found.
	Reasons	The code that sets the parameter value is too complex for analysis, or the compiler's optimization has made it so.
	Solution	See below, “compiling programs for RapiTime”, or contact Tidorum Ltd.

Compiling programs for RapiTime

The GCC compilers for SPARC (including the GNAT Ada compiler) perform some optimizations that can make it difficult for Bound-T to find the “name” for some RapiTime ipoint calls. The most common reason for this problem is a switch-case statement where every branch ends with an ipoint (a call of the ipoint subprogram) but with different “names” (different parameter values). The compiler may detect the call instructions (and their delay slots) as common code, shared by all case branches, and remove them from the branches in favour of a single call instruction (and its delay slot) to which all case branches jump. Bound-T sees this as a single ipoint that has multiple “names” (any “name” or parameter value defined in any case branch). This leads to the error message that the ipoint-name is unknown.

Experiments have shown that the following GCC compiler options disable the problematic optimizations and avoid this error:

```
-fno-optimize-sibling-calls
-fno-crossjumping
```

5 WRITING ASSERTIONS

If you use Bound-T to analyse non-trivial programs you nearly always have to write *assertions* to control and guide the analysis. The most common role of assertions is to set bounds on some aspects of the behaviour of the target program, for example bounds on loop iterations, that Bound-T cannot deduce automatically. Assertions must identify the relevant parts of the target program, for example subprograms and variables. The assertion language has a generic high-level syntax [17] in which some elements with target-specific syntax appear as the contents of quoted strings:

- subprogram names,
- code addresses and address offsets,
- variable names,
- data addresses and register names,
- instruction roles, and
- names of target-specific properties of program parts.

In practice the *names* (identifiers) of subprograms and variables are either identical to the names used in the source code, or some "mangled" form of the source-code identifiers where the mangling depends on the cross-compiler and not on Bound-T. However, Bound-T defines a target-specific way to write the *addresses* of code and data in assertions. *Register names* are considered a kind of "data address" and are target-specific.

This chapter explains any specific limitations and possibilities for user-specified assertions when Bound-T is used with SPARC programs. These issues include the identification of subprograms and variables by machine addresses, the name mangling in the GNAT and GCC compilers, and the SPARC-specific property names.

5.1 Naming Subprograms

Ada modules

In Ada modules, the naming is complicated by the package hierarchies, nested subprograms and overloaded names. With the ORK system using the GNU Ada compiler, the general principle is that an Ada identifier of the form *A.B.C* is mapped to a linker symbol of the form *a_ _b_ _c*. In other words:

- letters are converted to lower case,
- periods are converted to double underscores (_ _).

However, for an Ada entity that is a library-level subprogram, for example the main procedure of the Ada program, the identifier, say *Foo*, is mapped to the symbol *_ada_ _foo*. In other words:

- letters are converted to lower case,
- a prefix of one underscore, *ada* and a double underscore is added.

Section 4.3 explains the additional rules for naming tasks and protected objects.

Ada lets different subprograms have the same *overloaded* name when the compiler can distinguish the subprograms based on the parameter and result types (the *profile* of the subprogram). Linkers, however, need unique names, so GNAT will assign sequential

numbers to such overloaded names, in the order in which they are declared in the source code, and construct unique linkage names by appending a *$* symbol and the number to the overloaded name. For example, if package *Pak* contains two subprograms named *Foo*, the linkage name for the first one will be *pak_ _foo$1* and for the second, *pak_ _foo$2*.

Be very careful to update your assertions when you add, remove or reorder subprograms with overloaded names in a package. In the preceding example, if you insert a new overloaded subprogram *Foo* in package *Pak* between the two existing subprograms *Foo*, the new subprogram gets the linkage name *pak_ _foo$2* while the linkage name of the last subprogram changes to *pak_ _foo$3*. Existing assertions for the last subprogram have to be updated accordingly or they will be applied, wrongly, to the new, inserted subprogram.

For Ada source programs the subprogram linkage names can most easily be found from the program itself by using the Bound-T option *-trace symbols* to list all the symbols. Alternatively, the target program's symbol-table can be dumped with Bound-T or with some special-purpose program such as the GNU *objdump*.

C and Assembler modules

The ORK development tools do not include any extra underscores before the subprogram name for C and assembler sources. The identifier given in the source-code is used as such.

5.2 Naming Variables

The naming of the variables by the ORK tools is more straightforward. The scope of the variables includes the filename and the subprogram name, but no extra underscores are included in the variable names for Ada, C or assembler sources.

5.3 Naming Items by Address

Registers

SPARC registers can be named in assertions with the *address* keyword, followed by a quoted string that gives the register name.

The syntax for register names will appear rather strange to those familiar with the SPARC assembly language. The reason behind this syntax is the register-window system which means that a subprogram can refer to the same register-file location with two different register numbers – one number before the **SAVE** instruction and another number after the **SAVE**. Moreover, a given register number can identify a different location in the register file before and after the **SAVE**, and some register-file locations are accessible only before or after the **SAVE**.

Bound-T needs to name the registers in a way that associates a given name with the same storage location both before and after the **SAVE**. This is done by expanding the usual SPARC register groups – the “global”, “local”, “in” and “out” groups – to include two new groups: “pass” registers and “work” registers. These six groups cover all the registers that can be accessed both before and after the **SAVE**, as shown in the table below. The symbol *n* means any number in the range 0 .. 7 with some exceptions as noted in the table. The stack pointer and frame pointer registers are modelled separately as shown in the table.

Table 25: Register groups and names

Bound-T name	*Meaning*	*Corresponding SPARC registers*	
		Before SAVE	*After SAVE*
`Gn`	A global register. The dummy register `R0` is not modelled so the name `G0` is not used.	`G0..G7 = R0..R7`	
`In`	Input register as seen before the **`SAVE`** instruction (in the caller's view). The frame pointer is modelled separately so the name `I6` is not used.	`I0..I7 = R24..R31`	Not accessible
`Ln`	Local register as seen before the **`SAVE`** instruction (in the caller's view).	`L0..L7 = R16..R23`	Not accessible
`Pn`	Incoming parameter-passing register, accessible both before and after the **`SAVE`**.	`O0..O7 = R8..R15`	`I0..I7 = R24..R31`
`Wn`	Working register after the **`SAVE`**.	Not accessible	`L0..L7 = R16..R23`
On	Outgoing parameter register after the **`SAVE`**. The stack pointer is modelled separately so the name `O6` is not used.	Not accessible	`O0..O7 = R8..R15`
`SP`	The stack pointer.	`SP = R14`	
`FP`	The frame pointer.	`FP = R30`	

The syntax for ordinary register names is a one-letter register-set identifier, as shown in the leftmost column of Table 25, in upper or lower case, followed by a decimal register number within the valid range. The valid range depends on the register-set as explained in the table. The names for the stack pointer and frame pointer are the two-letter words SP and FP, respectively, in upper, lower or mixed case.

For example, the assertion

```
variable address "g5" <= 100;
```

states that the value of **`g5`** = **`r5`** is at most 100.

Variables

Variables can be named in assertions with the *address* keyword, followed by a quoted string that gives the variable's memory address. The memory address of a variable is given by an address-space identifier *m* or *M* followed by the hexadecimal address. The hexadecimal address is given by using decimal numbers 0 - 9, and letters a, b, c, d, e and f (case-insensitive). Note that the address must not be preceded by "0x" nor surrounded by "16# .. #" nor followed by an "h" suffix; just write the hexadecimal digits.

For example, the assertion

```
variable address "m57D12" 12 .. 20;
```

states that the value of the 32-bit integer variable at the address 57D12 (hexadecimal) is between 12 and 20.

Subprograms

Subprograms can be named in assertions with the *address* keyword followed by a quoted string that gives the entry address in hexadecimal as above (but not preceded by any specific identifier).

For example, the assertion

```
subprogram address "A70D21E"
   time 342 cycles;
end subprogram;
```

states that the WCET of the subprogram with the entry address A70D21E (hexadecimal) is at most 342 cycles.

5.4 Loop and Return Offsets

An assertion can identify a loop by giving an offset from the start of the subprogram that contains the loop, in the form

```
subprogram "Nurture"
   loop that executes offset "1A8"
      repeats 15 times;
   end loop;
end "Nurture";
```

For the SPARC the loop-offset is written as a quoted string that gives the entry address in hexadecimal (but not preceded by any specific identifier). The example above identifies the loop that contains the instruction at the address given by the entry address of the subprogram *Nurture* plus 1A8 (hexadecimal) octets. Note that the offset must point exactly at an instruction; the assertion will not work if the offset points, for example, to the second octet of an instruction. The offset that Bound-T displays for an unbounded loop should work as such.

Code offsets can also be used in "return to offset" assertions for subprograms that return in special ways. The same offset syntax is used there.

5.5 Instruction Roles

The generic assertion language [17] contains syntax for asserting the "role" that a given instruction (identified by its address or offset) performs in the computation, for example whether a branch instruction performs a branch or a call. The roles and their names are target-specific. The SPARC version of Bound-T defines no assertable roles; it chooses the role of each instruction based on its own analysis of the instruction and its context.

5.6 Properties

Assertable properties

The assertable properties for the SPARC are listed and explained in the following table. For the properties that relate to memory wait cycles, see Table 17 in section 4.2 for the conversion from wait states to wait cycles.

Table 26: Assertable Properties

Property name		*Meaning, values and default value*
read_ws	*Function*	Changes the number of read wait cycles in the current context, for all memory reads (excluding stack references, alternate memory references and code fetches).
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-read_ws*.
write_ws	*Function*	Changes the number of write wait cycles in the current context, for all memory writes (excluding stack references and alternate memory references).
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-write_ws*.
code_ws	*Function*	Changes the number of code fetch wait cycles in the current context.
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-code_ws*.
alt_read_ws	*Function*	Changes the number of read wait cycles in the current context for alternate memory references.
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-alt_read_ws*.
alt_write_ws	*Function*	Changes the number of write wait cycles in the current contex for alternate memory references.
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-alt_write_ws*.
stack_read_ws	*Function*	Changes the number of read wait cycles in the current context for stack references.
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-stack_read_ws*.
stack_write_ws	*Function*	Changes the number of write wait cycles in the current context for stack references.
	Values	Number of wait cycles (system clock cycles). See section 4.2 for the relation to wait states.
	Default	Zero wait cycles or the value given in a command-line option *-stack_write_ws*.
call	*Function*	Can be asserted for a subprogram and specifies the calling sequence. Since Bound-T deduces the calling sequence for non-root subprograms from the actual call instructions this property should be asserted only for the root subprogram of an analysis and only when this is not an ordinary subprogram called with the normal sequence.

Property name		*Meaning, values and default value*
	Values	0 Normal calling sequence, ordinary subprogram. 1 Alternate calling sequence, ordinary subprogram. 2 Trap handler. 3 Interrupt handler.
	Default	Zero (0) (normal calling sequence) for a root subprogram, otherwise deduced from the instructions used in the call(s) of the subprogram.
caller_save	*Function*	Can be asserted for a subprogram and specifies whether the subprogram is called with the **CALL**;**SAVE** sequence, that is, whether the caller executes the **SAVE** instruction that allocates a register window for the called subprogram, and if so, how much stack space this **SAVE** allocates.
	Values	Zero (0) or undefined means that the caller does not execute a **SAVE** on behalf of this subprogam. A positive value means that the caller executes a **SAVE** for this subprogram; the value gives the amount of stack space (in octets) that this **SAVE** allocates. A positive value should not be asserted for subprograms that are trap or interrupt handlers.
	Default	Zero (0) for root subprograms, otherwise deduced from the instructions used in the call(s) of the subprogram.
win_depth	*Function*	Asserts the range of the window-depth of the current subprogram (given in subprogram context).
	Values	The number of register windows used by this subprogram and its deeper-level callees.
	Default	The range of window-depth of a subprogram is analysed by Bound-T, or if the WCET of the subprogram is asserted and the subprogram is not analysed at all, a default value of zero will be used.
rwu_max	*Function*	The register window usage assumed on the entry to the current subprogram (given in subprogram context).
	Values	Number of register windows.
	Default	Two (2) or the maximum number of windows (*-max_win* option) if less than 2, for the root subprograms named on the command-line, and the result of the analysis for other subprograms.

Properties assumed for unanalysed subprograms

When a subprogram is excluded from the analysis by asserting its WCET, the subprogram is given default values for properties as shown in the following table. Other values for these properties can be asserted to override these defaults. The table lists only the properties that are relevant; for example, the memory wait-state properties are irrelevant because the WCET of the subprogram is asserted.

Table 27: Default properties for unanalysed subprograms

Property name	*Default value for a subprogram that is not analysed*
call	Undefined. Can be asserted or deduced from the actual calling sequence.
caller_save	Undefined. Can be asserted or deduced from the actual calling sequence.
win_depth	0 .. 0.

Property name	*Default value for a subprogram that is not analysed*
rwu_max	Two (2) or the maximum number of windows (*-max_win* option) if less than 2.

Tidorum Ltd

Tiirasaarentie 32
FI-00200 Helsinki, Finland
www.tidorum.fi
Tel. +358 (0) 40 563 9186
VAT FI 18688130