Compiler Support for WCET Analysis: a Wish List

G. Bernat
Real-Time Systems Research Group
University of York, England, UK
bernat@cs.york.ac.uk

N. Holsti
Space Systems Finland Ltd
Espoo, Finland
niklas.holsti@iki.fi

Abstract

Static timing analysis of a computer program needs both high-level information from the source code of the program, and low-level information from the compiled object code. Compilers and linkers could support such analysis by providing more and better information about the structure and behaviour of the source and object code and about the relationship between source and object code. Moreover, some parts of timing analysis would be eased by more control over the code generation process. Finally, timing analysis often depends on annotations or assertions embedded in the source code, or referring to the source code. Compilers and linkers could help us make use of annotations by translating the annotations from the source domain to the object domain.

To make these needs known to compiler developers and vendors, we propose the collection of a "wish list" of requirements from academic and industrial groups working in timing analysis. We discuss how such a list should be selected, presented and motivated, with emphasis on finding other users with similar needs, for example other kinds of static analysis, debugging or program verification.

1. Introduction

Static analysis of a program's timing behaviour, such as WCET analysis, needs both high-level information from the source code of the program, and low-level information from the compiled and perhaps even linked machine code. Some parts of WCET analysis are easier on the source-code level, for example path analysis and pointer analysis, but obviously the actual machine code must be analyzed to find the actual execution time. Conversely, static analysis of the machine code alone can be quite difficult. For example, if the compiler has generated branch instructions with dynamically computed target addresses, it is hard to build the machine-level control-flow graph although the source-level control-flow may be quite static and simple, such as a switch/case statement.

The information on the high and low levels must also be correlated, for example to find the correspondence between source-code control flow and machine-code branches, or between source-code variables and machine registers or memory locations.

Since the compiler and linker generate the machine code from the source code, they are best placed to create the correlation between the two levels, and to some extent already do so by emitting debugging information such as symbol tables and memory maps. Present-day compilers and linkers also perform quite a lot of program analysis themselves, but usually do not make the results available for other tools such as static code analysers and WCET analyzers. WCET analysis tools have to reconstruct this information from source code and object code alone; this is a challenging task and sometimes difficult to perform. WCET researchers and tool developers would often like more and better information and support from the compiler and linker. At WCET 2002 it was proposed the creation of a "wish list" of better compiler support for WCET analysis.

The purpose of this paper is to collect the requirements from the WCET analysis community from individual efforts with the long term objective of influencing tool manufacturers (specially compiler vendors) to generate intermediate data formats useful for timing analysis.

2. The role of compilers in WCET analysis

The kind of WCET tools we are considering are those that analyze machine code (or code in a low-level intermediate language) and perhaps also source code, but are not integrated with a compiler. For correlating the source-code with machine code the tools must thus depend on the additional information generated by the target compiler and linker, whch is usually just the debugging information. This information is often insufficient and creates unnecessary problems for the WCET analysis.

Some WCET researchers have modified existing compilers or built their own compilers and even new programming languages with better support for WCET analysis. They have studied how a compiler can generate useful information and also how this information, derived from the source-code, can be maintained and translated through the compilation and linking process to apply to the machine code. This is a valid research area but we feel that it is unlikely to yield useful production compilers with WCET support. We believe that most WCET R&D groups, and certainly most software developers who are potential users of WCET tools, would prefer to use the common target languages, compilers and linkers, mainly for customer support and certification issues. This reduces the threshold for users to adopt WCET tools but it requires us to persuade the compiler and linker suppliers to change their tools to support WCET analysis better. The experience from the development of special languages and special compilers with support for WCET analysis will be useful here.

The ultimate goal is the definition of a standard format of code transformations and code properties that is produced by compiler tools. The standardisation would allow the seamless integration of this data across tool chains.

3. The clients of the compiler and linker

The proposed wish-list for improved compiler and linker support must of course serve the needs of WCET analysis, but to make the list persuasive, we should use the fact that there are many other users (clients) of the outputs from the compiler and linker. The target processor that executes the machine code is only of these users, which include at least:

- The linker (as a client of the compiler and of a previous run of the linker),
- The loader (as a client of the compiler and linker),
- The compiler itself, in several possible ways: separate compilation of module specifications and header files; interfaces between compiler passes; use of run-time monitoring results for optimization, etc.,
- The debugger (tool and human) and disassembler,
- Machine-code instrumentation, translation and verification tools,
- Manual machine-code review and tools to support such review,
- The target program itself, for reflection or intro-spection purposes such as exception handling, stack unwinding, garbage collection, run-time verification, etc.,
- Other code analysis tools, including profilers, memory usage analysis tools, static code analysers (for example like Spark), etc
- And, last but in our view not least, WCET analysis tools.

Any wish for added WCET support is more likely to be implemented if it benefits other clients, too. Such *collateral benefits* should be actively sought and clearly presented.

4. General guidelines

We invited the WCET03 workshop to discuss and collect a list of requirements from the community. We have initially classified the set of requirements according to the following categories:

- Properties of source code level: Including tree structure of the code, implicit type conversions, results of pointer analysis, dead code analysis, variable sizes of arrays, value-range analysis, loop induction variable analysis, annotations, type analysis and range analysis for automatic deduction of ranges of loop bounds, multiple language support, Virtual method invocations in OO languages, etc..
- Properties of machine code. For example, the list of the possible targets of a dynamic branch instruction that corresponds to a switch/case statement.
- Mapping between the source-code, intermediate code and machine-code levels. For example, the location in the machine code that corresponds to a WCET annotation in the source. Automatic extraction of code annotations.
- Map of code transformations, mostly code optimi-sations so that one-to-one mappings between source code and object code can be derived.
- New compiler controls or options to make the machine code easier to analyze. For example, special restrictions on optimization such as creating irreducible loops.
- New functionality for the compiler tool chain: automatic instrumentation of programs for coverage analysis and for timing instrumentation.
- Standard ways to annotate real-time and WCET aspects in the source code, with translation to the machine-code level. For example, annotations for loop bounds and path constraints.

For each wish, the list should explain clearly what is desired (taking into account that the audience are not WCET analysis experts) and why it will be useful to WCET analysis *and other tools*. The list should suggest

how the wish could be implemented in a compiler or linker, with reference to any existing implementation in a research context. For new information to be provided by the compiler or linker, the list should suggest the format and medium, for example how the information could be encoded in ELF or DWARF or in a separate file. All information about the machine code should also be traced back to the source code.

5. Discussion

At the WCET03 workshop, we started the discussion on these requirements recording current developments by individuals, as well as desired functionality. The issues discussed include:

- What information is required from compilation tools
- Format of such information
- New compiler functionality for WCET analysis.

6. Conclusions

To enable WCET analysis detailed information already available in compiler tool chains is required. We propose to collect an agreed set of requirements from the WCET Community on the information needed to perform WCET analysis with the aim of producing a white paper to influence compiler manufacturers and vendors to make such information available.

Table 1, below, lists the items we have collected so far. ordered by requirement category. This list is of course not yet complete, and also the columns "Examples" and "Supported analyses" are incomplete. The authors would be most grateful for comments on this list and suggested additions to this list, dealing with the issues raised in this paper. Other future work includes finding collateral benefits, prioritizing the requirements, and defining the data formats and other interfaces for implementing the requirements.

Table 1. Compiler and linker support for timing analysis

Requirement category	*Property, mapping or control*	*Examples*	*Supported analyses*
Properties on source-code level	Tree structure of the code	Intra-procedural control structures: sequence, conditional, switch/case, loop, exception. Inter-procedural control structures: call, return (normal/alternate), exception.	Control flow.
	Implicit type conversions	Address to or from integer. Integer to long.	Values and arithmetic. Loop bounds.
	Types and value ranges or value sets of variables and expressions	Range of loop counters. Range of actual parameters. Pointer analysis results ("points-to" properties).	Feasible paths, loop bounds. Cache timing (memory access patterns, dynamic addresses).
	Array sizes	Dynamically created (heap) arrays. Local (stack) arrays with dynamic size. Formal array parameters to subprograms where actual parameter determines size.	Loop bounds. Cache timing. Stack usage bounds.
	Loop induction variables	Loop counters. Index expressions that depend on iteration count.	Loop bounds. Cache timing.
	Content and location of source-code annotations	Loop-bound annotations. Memory timing annotations.	Potentially all.

Requirement category	*Property, mapping or control*	*Examples*	*Supported analyses*
	Feasible paths and loop bounds (as deduced by compiler).	Dead code detected e.g. by constant propagation. Loop bounds for compiler-generated loops (e.g. copying loops).	Control flow, feasible paths, loop bounds.
Mapping source code to object code	Source lines to code instructions	As currently implemented in "debug" information.	Support other mappings, e.g. mapping of path constraints or annotations.
	Source tree to code instructions and branches	Altered order of then/else in conditional statement. Altered order of cases in case/switch statement. Changes in the placement of a loop termination test (test at start/middle/end of loop). Other loop transformations, unrolling etc..	Support other mappings.
	Source annotations to code	Map a loop-bound annotation to the loop (head) in the object code. Adapt a loop-bound annotation to the transformations applied to the loop.	Loop bounds.
Properties on object code level	Possible targets of dynamic branches.	Switch/case structures implemented with jump tables or address tables	Intra-procedural control flow.
	Possible callees for dynamic calls.	Late-bound method calls in object-oriented programming. Interrupt handlers and trap handlers, called via vector tables.	Inter-procedural control-flow.
	Code that violates target-processor standards and needs special analysis	Library routines or compiler-generated routines that have non-standard calling sequences.	Control-flow and others.
	How target-processor standards are used, when there are alternatives	For each subprogram or call: which of the alternative calling sequences and parameter-passing methods is used.	Any aspect of analysis influenced by target-processor standards, for example inter-procedural control flow and data flow.
	Logical role of multi-purpose instructions	Whether an instruction that loads the Program Counter represents a jump, call or return.	Control flow and others.
	Operand type information for polymorphic instructions.	Signed versus unsigned interpretation of integer arithmetic and comparison instructions and of immediate (literal) operands.	Values and arithmetic. Feasible paths, loop bounds, pointers.
	Logical effect of code subsequences	On processors with small word size, e.g. 8 bits, the fact that a certain sequence of 8-bit computations has the effect of adding two 16-bit quantities.	Values and arithmetic. Feasible paths, loop bounds, pointers.

Requirement category	*Property, mapping or control*	*Examples*	*Supported analyses*
	Code that relies on over-flow or other exceptions for nominal operation	A loop from 0 to 255 using an 8-bit counter might rely on overflow from 255 to 0 in the last iteration.	Values and arithmetic. Loop bounds.
	Memory locations that are initialized dynami-cally at program start but are constant during run.	Trap vector tables. Constants copied from PROM to RAM.	Values and arithmetic. Control-flow analysis when the values enter dynamic branch or call computations. Feasible paths, loop bounds, pointers.
	Memory locations with special semantics	Volatile variables (consecutive reads may give different values). Control registers with different read/write roles (a read does not return the last written value).	Values and arithmetic. Feasible paths, loop bounds.
Control over object code generation	Control the generated loop structures	Generate only reducible loops. Prevent loop unrolling or other specific loop transformations.	Loop analysis, loop bounds. Support source-to-object mappings.
	Control the generated inter-procedural transfers	Prevent or enforce inlining. Enforce target-standard procedure calling protocols.	Inter-procedural control-flow. Support source-to-object mappings.