Status of the Bound-T WCET Tool

Niklas Holsti and Sami Saarinen
Space Systems Finland Ltd
Niklas.Holsti@ssf.fi, Sami.Saarinen@ssf.fi

Abstract

Bound-T is a tool for static WCET analysis from binary executable code. We describe the general structure of the tool and some specific difficulties met in the analysis of the supported processors, which are the Intel 8051 8-bit microcontrollers, the Analog Devices ADSP-21020 Digital Signal Processor, and the SPARC V7 processor. For the DSP, the problem is the complex program sequencing logic using an instruction pipe-line and nested zero-overhead loops with implicit counters and branches. The solution is to model the full sequencing state in the control-flow graph. For the SPARC, the problems are the register-file overflow and underflow traps, which may occur at calls and returns, and the concurrency of integer and floating-point operations, which may force the Integer Unit to wait when it interacts with the Floating-Point Unit. The traps are modelled with a whole-program analysis. The IU/FPU concurrency is modelled by distributing the potential waiting times onto flow-graph edges in a heuristically optimal way, also using some inter-procedural analysis.

1 Introduction

The Bound-T tool analyses compiled and linked executables to find the WCET, flow graphs, call graphs, and stack usage. Space Systems Finland (SSF) developed the tool with support from the European Space Agency (ESA) for space applications [1][2][3]. SSF is developing the tool further, aiming also at non-space applications.

The target processors currently supported are the Analog Devices ADSP-21020 (a 32-bit floating-point DSP architecture, forerunner of the SHARC), the Intel 8051 (a large family of 8-bit microcontrollers), and the SPARC V7 (a 32-bit RISC general-purpose architecture). All these processors are used in ESA space projects, which is one reason why they were chosen for Bound-T. The other reason was to implement some quite different architectures in order to verify the adaptability of the tool design. Since Bound-T uses only binary code, it is independent of the source language of the program to be analysed.

The most advanced feature of Bound-T is the automatic analysis of loop-bounds, using a model of the program's arithmetic. The same analysis provides some context-sensitivity by propagating actual parameter values into the analysis of the called subprogram. To supplement the automatic analysis, the user may state assertions on loop bounds, variable ranges, and other useful facts.

The main limitations of Bound-T are currently the lack of analysis of cache memories, a limited analysis of aliasing and dynamic branching, and the difficulty of high-level analysis based on low-level code, especially if the machine word is short, for example a loop with a 16-bit counter running on an 8-bit machine. The arithmetic analysis for loop bounds is occasionally very time-consuming and sensitive to the structure of the program.

The rest of this paper paper is organized as follows. Section 2 discusses the architecture of the tool. Section 3 describes the modelling of the target processor and target program thru abstract data types. Section 4 presents the major analysis phases and methods. Sections 5 and 6 focus on some interesting and difficult problems: the control-flow analysis of the ADSP-21020, and the IU/FPU concurrency in the SPARC. Section 7 reports on the commercialization and section 8 sketches future work.

2 Tool architecture

Bound-T is based on target-specific modules for reading and decoding binary files, generic modules for creating the control-flow graphs and call-graphs, a Presburger Arithmetic package (*Omega*) [4] for modelling the arithmetic of loop-counters, and an Integer Linear Programming tool (*lp_solve*) [5] to find the worst-case path.

The architecture of Bound-T was designed to be *adaptable* to different target processors, *extensible* with new kinds and methods of analysis, and *portable* to different host platforms.

The easy part of adaptability is to isolate the target-dependent parts into target-specific modules. The hard part is to make the interface of these modules valid for all targets. Our approach is to abstract the important aspects of the target processor. This has worked well, as shown by the range of supported targets.

Extensibility is provided in the conventional way by dividing the analysis into phases, with the result of each phase stored in the program-model. This method is limited by the fact that the data structures of the program model are hard-coded (as opposed to a data-base, for example). However, there are hooks to target-specific data and operations which have let us implement the new SPARC analyses in the Bound-T framework.

For portability, we use a portable implementation language (Ada). The current user interface, based on the command-line and text inputs, is trivially portable.

3 Processor and program models

3.1 Processor model

The target processor is modelled by several abstract data types. The most important type is the identifier or *address* of a control-flow *step*. For a simple processor like the Intel 8051, a step-address is just the address of an instruction. For a processor with complex program-sequencing, a step-address can contain much more context (see section 5 for the ADSP-21020 example). The step-address type is the basis for creating the control-flow graphs, where nodes are identified by a step-address.

Another important abstract type models the registers, flags and memories of the processor, or in general any *cell* that can store an integer value. The cell type is the basis for modelling the arithmetic computations and branching conditions. Some cells are just an enumeration, for example all the processor registers that are always statically addressed. Other cells can represent storage addressed in complex or dynamic ways, for example parameters accessed relative to the stack pointer.

These two abstract types divide our model of program state into a *control state* (step-address), modelled by the control-flow graph, and a *data state* (values of cells), modelled by Presburger input-output relations.

Our processor-model is essentially limited to single-threaded processors, although synchronized internal concurrency is possible. In other words, there may be several functional units running concurrently, as long as they all execute the same instruction stream as in VLIW machines. In the SPARC, the IU and FPU are not that strictly synchronized, and so this processor is in principle out of scope for our model.

3.2 Subprogram model

A subprogram under analysis is represented by a control-flow graph (CFG). A node in the CFG is a basic block and contains a sequence of steps. A step usually corresponds to a machine instruction, but in special cases instructions may be split into several steps, or consecutive instructions may be bundled into one step. For example, the Intel-8051 may use two consecutive 8-bit immediate-load instructions to load an immediate 16-bit value into the 16-bit Data Pointer register, but the arithmetic analysis is easier if the two 8-bit instructions are decoded into one 16-bit load-step in the CFG.

Calls to other subprograms are represented by special CFG nodes (steps) that refer to the callee.

3.3 Timing model

Each step in a CFG has an associated worst-case execution *effort* which depends mainly on the instruction(s) the step represents, but can also depend on the context (via the step address). The effort is a target-specific abstract type, as are the types for the total *work* to execute a sequence of steps and edges, and the processor *power* that determines the number of processor cycles taken by some amount of work. Memory accesses and wait-states are modelled in the effort, work, and power.

Each edge in a CFG has an associated worst-case execution time which typically models two things: (1) the extra time taken to actually branch from a conditional branch instruction, and (2) interference between consecutive instructions. For example, when an ADSP-21020 instruction that uses an Address Generator unit is immediately preceded by a load-register into this address generator, it takes two cycles instead of one cycle. We assign the extra cycle to the edge between the two instructions.

Branch delays due to instruction pipe-lining, where a branch takes effect only after some “delay slot” cycles, are modelled in the step-address (sequencer model), not in the execution time of the branch edge.

Since we use ILP to find the worst-case path in a CFG, we assume that the total execution time of a path is at most the sum of the times for the nodes and edges on the path.

3.4 Arithmetic model

The arithmetic effect of a step is represented by a set of assignments of expressions to cells. The expressions are Presburger formulas, possibly conditional, operating on constants and cell values. Thus, an unconditional formula is a sum of terms where each term is a constant or a cell or the product of a constant and a cell. A conditional formula chooses one of two such sums depending on a Presburger condition, which is a comparison between two Presburger formulas. It is also possible to state that a cell is set to an unknown value.

For example, consider a step (an instruction) that increments the register A and sets the Z flag if the result is zero. The effect is represented by the two assignments $A' = A + 1$ and $Z' = \textit{if } A+1 = 0 \textit{ then } 1 \textit{ else } 0$, where a prime indicates the new value of a cell, so A' = new value of A.

The Presburger formalism could in fact model any Presburger relationship between the "before" and "after" values of the cells, for example $A = A' - 1$. We use only the functional form (new value = function of old values) to allow other analyses such as *def-use* chaining or constant propagation.

Each CFG edge has a precondition that is a Presburger formula that must be true when this edge is executed. Thus, the precondition is a *necessary* condition for executing the edge, but perhaps not a *sufficient* one. For example, consider a step that decrements register B, jumps to another step if the result is not zero, and otherwise continues to the next step in address order. The edge to the next step has the precondition $B = 0$ while the edge for the jump has the precondition $B \neq 0$. A precondition that is unknown (or too complex to be analysed) is represented by the constant *true*.

Since our aim is only to find loop bounds (not, for example, numerical errors such as division by zero), we only model those instructions and storage-cells that are likely to be used for loop counters. For floating-point instructions we only model the side-effects on the integer computation. On the ADSP-21020 for example, where any 32-bit general register can be used for integer computation or floating-point computation, a floating-point computation is considered to yield an unknown register value.

We generally assume that the computation does not overflow or underflow. For the 8-bit Intel-8051 we provide a command-line option to negate this assumption. For example, if register A in the above example is 8 bits wide, unsigned arithmetic is used and overflow is considered, the effect of incrementing A would be encoded as $A' = \textit{if } A = 255 \textit{ then } 0 \textit{ else } A + 1$. Unfortunately this creates many conditional assignments which slows down the Presburger solver markedly.

Note that we use a *symbolic* model of the arithmetic; we do not *simulate* the arithmetic by actually computing expressions and assigning their values to simulated cells. The analysis is static and based on solving or simplifying the system of equations and constraints that represents the joint arithmetic effect of all the analysed instructions.

3.5 Program model

The program model consists of all the subprograms under analysis, starting from the "root" subprograms named by the user and including all their callees. We will use the term "call" to mean a specific step, in the CFG of a caller subprogram, that represents a call to a specific callee subprogram.

Each call is associated with a parameter-passing map between the cells in the caller and the cells in the callee. This map is used to propagate bounds on parameter values from the caller to the callee, perhaps for several call levels, in the hope that these parameters define loop bounds. We do not track the other data-flow direction, from callee to caller. The value of any cell that is modified in the callee is considered unknown after the return to the caller.

4 Analysis phases and methods

4.1 Overview

The analysis phases in Bound-T are, in order:

1. Reading the target program.
2. Instruction decoding and control-flow tracing.
3. Arithmetic analysis for dynamic branches.
4. Arithmetic analysis for dynamic data accesses.
5. Loop bounding analysis.
6. ILP analysis to find the worst-case path.

After reading in the binary program and its symbol tables, Bound-T traces the control-flow starting at the root subprogram entry-points. Each instruction is decoded, entered in the CFG as a step (or many steps, or part of a step), and the possible successors (new step addresses) are found and decoded in turn. When a call instruction is found, the callee is added to the set of subprograms to be analysed. This phase terminates when all paths end with a return instruction (a step with no successors) or a dynamic branch (successors so far unknown).

For subprograms with dynamic branches, arithmetic analysis is applied to try to resolve the target addresses of each branch. If this succeeds, the exploration of the control flow is resumed from these addresses. There may be several iterations of flow-analysis and arithmetic analysis.

When the CFGs of all subprograms are complete, arithmetic analysis is applied to try to resolve dynamic data accesses. Unresolved accesses are left as such, and are considered to yield unknown values.

Loop bounding analysis tries to find cells that act as loop counters, to find bounds on the values of these counters, and thus to bound the number of loop iterations.

When loop-bounds in a subprogram are known, we use the Implicit Path Enumeration Technique [6] to find the worst-case path in the subprogram as the solution of an ILP problem where the unknowns are the number of times each CFG node or edge is executed, the constraints are derived from the CFG and the loop-bounds, and the objective is to maximise the total execution time of the subprogram. Subprograms are processed in bottom-up order so that the WCET of each call is known when the caller is processed.

The following sections explain the arithmetic analysis and the loop bounding analysis in more detail. The other phases use conventional methods.

4.2 Arithmetic analysis

In the arithmetic analysis of a subprogram, the arithmetic effects of several steps in the CFG are joined to give the overall effect of some execution path. In mathematical terms, the effect of a step is a relation between the cell values before and after the step, called the *input-output relation*. The input-output relation for a sequence of steps (a path) is computed simply by joining (chaining) the relations of the steps. The preconditions on edges in the path are included as additional constraints in the chain. In an acyclic part of a CFG, a simple one-pass algorithm can compute the input-output relation between any pair of steps. When there are several paths between the steps, the "incoming" relations to a step where the paths join are combined by set union (disjunction).

Loops are handled by a bottom-up traversal. The body of an innermost loop is acyclic, so we can compute the input-output relation of the body. From this relation, we find the cells that must be loop-invariant (output value = input value). The entire loop is then fused into one step with an effect that approximates the effect of the loop as an input-output relation that keeps the loop-invariant cells constant and does not constrain the other cells, leaving them with unknown values. (To be precise, the relation does include the constraints created by the loop-termination paths, from the loop-head to a loop-exit, assuming unknown values at the loop-head.) The next higher (containing) level of loops can then be analysed, fused and approximated in the same way.

In some target processors every instruction sets many flags. This creates many conditional assignments in the effect of the instruction, but most of these are "dead" because the flag is redefined by another instruction before it is used. Before the actual arithmetic analysis as described above, we do a live-variable analysis in the normal way, and include only the live assignments in the input-output relations.

The results of this arithmetic analysis are the input-output relations from the subprogram entry-point to each step in the subprogram, or along other interesting paths. From these relations we compute bounds on the values of cells at specific steps, for example bounds on the addresses of dynamic branches and data accesses, or bounds on the actual parameters in calls to lower-level subprograms. Derived or asserted bounds on the parameters of this subprogram, or on local or global variables, can define or sharpen the computed bounds.

4.3 Loop bounding analysis

Loops are bounded with a "syntactic" method, to use Gustafsson's terminology [7]. This method is faster than the abstract interpretation method proposed in [7] but applies only to counted loops.

Each loop is analysed separately as follows, in top-down and flow order. Let the *repeat relation* be the input-output relation that represents repetition of the loop, in other words all paths from the loop-head through the loop-body back to the loop-head, with all inner loops approximated as described in the preceding section. All the non-invariant cells in the repeat relation are candidates for loop counters. For each candidate cell, we compute bounds on the candidate's *initial value* before the loop, on the *repeat value* implied by the repeat relation, and on the change in the value implied by the repeat relation. Consider for example this Ada loop:

```
j := 3;
loop
   j := j + 2;
   if ... then j := j + 3; end if;
   exit when j > 9;
end loop;
```

For the cell *j*, the initial value is strongly bounded to $j = 3$, the repeat value is bounded by $j \le 9$, and the change is bounded to $2 \le \Delta j \le 5$. Together, these imply that *j* is an up-counter and that the loop-head can be re-entered from the loop-body at most *ceil* $(9 - 3 + 1) / 2) - 1 = 3$ times, for a total of 4 iterations of the loop.

In this way we can discover up-counters and down-counters. By computing the complement (negation) of the repeat relation we can discover counters that terminate the loop on equality ("exit when $j = 9$"), but only if Δj is exactly 1.

5 ADSP-21020 program sequencing

Digital Signal Processors often have special support for loops, to speed up vector computation and digital filtering. The ADSP-21020 has an instruction of the form "DO address UNTIL condition" which dynamically creates a loop that starts at the next instruction and ends at the given address. The DO UNTIL instruction sets the processor into a state where fetching the instruction at the given address makes the processor decide whether to repeat or terminate the loop at this address. In other words, from this address control may either continue onwards, or loop back to the instruction after the DO UNTIL, depending on the value of the condition flag two cycles earlier (due to pipe-line lag). Such loops can be nested to six levels and can interact in interesting ways with the delayed branch instructions.

For the ADSP-21020, we model in the step-address type the entire three-stage instruction pipe-line (fetch, decode, execute) and the whole six-level loop-nest. Since CFG nodes are labeled by step-address, a CFG edge now represents a transition from one specific pipe-line and looping state to another such state, so the CFG can very precisely model these transitions and their execution time.

An interesting side-effect is that a single instruction can appear as several steps in a CFG. For example, an instruction that follows a conditional delayed branch is decoded twice and represented as two steps, because the step-address for the branch instruction has two successor step-addresses, one that represents the case where the branch will be taken, and the other the case where it will not be taken. These step-addresses differ in the “fetch” stage of the pipe-line model. The delayed branch instruction is thus converted into an immediate branch in the CFG.

For another example, consider a block of instructions that can be entered either through a DO UNTIL instruction, or directly without a DO UNTIL. This whole block is then decoded twice, once in a context with an active loop-level for this DO UNTIL, and once without. However, this case is unlikely to occur in a real program.

6 SPARC IU/FPU concurrency

In the past year and with ESA support, we completed targeting Bound-T to the SPARC V7, including analysis of the register-file overflow and underflow traps and of the concurrency between the Integer Unit (IU) and the Floating Point Unit (FPU). In this paper, we focus on the IU/FPU concurrency.

6.1 The problem

In the SPARC V7, the IU is responsible for fetching all instructions and for executing integer instructions. Each floating-point (FP) instruction is forwarded to the FPU which executes the instruction while the IU fetches and executes new integer instructions. When a new FP instruction is found, but the FPU is still busy, the IU must wait until the FPU has finished the first FP instruction. The delay depends on the execution time of the first FP instruction, which increases the delay, and on the amount of IU computation between the two FP instructions, which decreases the delay. In the worst case, the delay can be nearly 80 cycles. An integer instruction is typically executed in one or two cycles.

One difficult aspect of this problem is that the delay depends on the path taken by the IU between the two FP instructions; this can be a large number (up to 80) of integer instructions. The delay is not an interaction between two *consecutive* instructions. Another difficult aspect is that the integer-execution time appears with a negative sign in the expression for the delay, which means that we would need the *best*-case IU time for computing the *worst*-case delay.

6.2 Finding delayed paths

The first phase in the IU/FPU analysis is thus to find the (potentially) *delayed paths* between two FP instructions (or from an FP instruction to itself), where the intervening integer computation is too brief to ensure that the first FP instruction has finished before the second one should start. Delayed paths are found simply by depth-first CFG traversals starting at each FP instruction and propagating the maximum remaining FPU-busy time along all paths, until another FP instruction is found or the FPU is sure to have finished the first FP instruction.

We avoid the need for best-case IU times by a principle we call “hurry up and wait”. If the IU executes the path between the two FP instructions *faster* than the worst-case bound, it will just have to wait *longer* for the FPU to finish the first FP instruction. In fact, the WCET of the first FP instruction is a worst-case estimate for the whole path from the start of the first FP instruction to the start of the second FP instruction, including the FPU-busy delay before starting the second FP instruction. Thus, we phrase the analysis in terms of the total time for each delayed path, not in terms of the actual delays.

6.3 Assigning delays to edges

Our objective is to minimize the pessimism, so that the FPU-busy delay is associated only or mainly with the paths that really incur delay in execution. The simplest approach would be to assign the worst-case delay time to each edge that enters the second FP instruction and that is part of a delayed path. This is pessimistic if this edge is part of the overall worst-case path and the worst-case path is not in fact delayed at this FP instruction, for example because it does not execute the first FP instruction. It could be less pessimistic to assign the delay to an edge that leaves the first FP instruction, if this edge is used only by the delayed path.

To avoid such pessimism, we use an ILP approach to distribute the worst-case blocking delay from *all* delayed paths onto *all* the edges in delayed paths so that the added pessimism is minimized (using a heuristic goal function, however). In this ILP problem, the unknowns are the additional delays to be assigned to the edges on delayed paths; the constraints are that the total execution time (including these additional delays) of every delayed path must be at least the WCET of the FP instruction at the start of the path; and the objective is to minimize an expression that estimates the pessimism.

6.4 Is it cheating?

When the FPU-busy delays have been distributed to the edges of delayed paths, the subprogram timing model — the CFG with times assigned to nodes and edges — is no longer a worst-case model, since the time assigned to the last edge and node of a delayed path is not necessarily an upper bound on the actual execution time of this edge and node. However, we can prove that the normal Bound-T WCET analysis (which assumes a worst-case model) still gives an upper bound on the execution of the whole subprogram, thanks to the constraints imposed on the distribution of the delays.

6.5 Inter-procedural aspects

To handle FPU-busy delays between FP operation pairs that cross subprograms, a bottom-up inter-procedural analysis assigns each subprogram a minimum *margin* of pure IU execution time on entry (before the first FP instruction is reached), and a maximum *legacy* of remaining FPU execution time on return (due to the last started FP instruction). These summary values are associated with the call steps in the analysis of the calling subprograms.

7 Marketing and commercialization

To commercialize Bound-T we have contacted a number of potential development partners, tool distributors and tool users, but the progress is very slow. WCET analysis is still poorly known, and it is often hard to make people understand what it does and how it differs from debugging, simulation and testing.

Partly because of the marketing delays, and partly because of staff shortages, no technical development is currently under way. SSF will itself use the SPARC V7 version in a major project that develops the on-board platform software for the ESA GOCE satellite (Gravity and Ocean Circulation Explorer). This will test the specific SPARC analysis methods described above, as well as the generic abilities of the tool.

We still hope to make Bound-T a commercially available tool, but the near-term plans are vague and depend on finding partners or users. We will gladly supply evaluation copies of Bound-T. The host platforms are Sun Solaris, Intel Linux, and Intel Windows (using CygWin and a command-line interface).

8 Future work

It is evident that the range of target processors should be increased and updated. Candidates for new targets include ARM, MIPS, AVR, and other microcontrollers.

We also have plans for several generic technical improvements. The derived loop-bounds could be used to sharpen the resolution of dynamic data accesses and dynamic branching, and the latter two should be iterated when needed, since some dynamic branching depends on dynamic data addressing (switch tables).

In the loop-bound analysis, it would be better to compute bounds on the difference between the initial value and the repeat value of a counter cell, instead of separate bounds on the two values. This would let us bound loops of the form “for *j* in *n* .. *n* + 10” even when the value of *n* cannot be bounded statically.

Nested loops where the bounds of the inner loop depend on the counter of the outer loop will currently yield pessimistic WCETs, because the worst-case bounds on the inner loop are assumed to hold for all executions of the outer loop. We don’t have a clear idea how this analysis could be improved in the Presburger method.

Better aliasing analysis and optional levels of aliasing analysis will probably be necessary for large target programs. Finally, while the current command-line interface is quite workable, a GUI would be convenient for browsing the analysis results of large programs.

9 References

[1] ESTEC/Contract No. 13362/77/NL/FM, “DSP Execution Time Estimation”.

[2] N. Holsti, T. Långbacka and S. Saarinen, “Worst-Case Execution Time Analysis for Digital Signal Processors”, *X European Signal Processing Conference*, EUSIPCO 2000.

[3] N. Holsti, T. Långbacka and S. Saarinen, “Using a Worst-Case Execution Time Tool for Real-Time Verification of the DEBIE Software”, *Proceedings of the DASIA 2000 (Data Systems in Aerospace) Conference* (ESA SP-457, ISBN 92-9092-669-4, September 2000), pp. 307-312.

[4] W. Pugh et. al., The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs, University of Maryland, http://www.cs.umd.edu/projects/omega.

[5] M. Berkelaar, ftp://ftp.ics.ele.tue.nl/pub/lp_solve.

[6] Y-T. S. Li and S. Malik, “Performance Analysis of Embedded Software Using Implicit Path Enumeration”. In *Proc. of the 32:nd ACM IEEE Design Automation Conference (DAC’95)*, 1995, pp. 456-461.

[7] J. Gustafsson, *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*, Uppsala University (Ph.D. Thesis, DoCS 00/115, ISSN 0283-0574) and Mälardalen University (MRTC 00/10, ISSN 1404-3041), May 200.