The Unthinkable: Automated Theorem Provers For (Tracing) Just-in-Time Compilers

Nikolai Tillmann Michał Moskal Wolfram Schulte Herman Venter Manuel Fahndrich

Microsoft Research, Redmond WA, USA

{nikolait,micmo,schulte,hermanv,maf}@microsoft.com

Abstract

Tracing just-in-time compilers (TJITs) determine frequently executed traces (hot paths and loops) at run time. These traces are then analyzed and optimized, and finally specialized machine code is generated. Up to now, TJITs employed standard compiler construction algorithms to analyze and optimize traces. We propose to leverage automated theorem provers to optimize traces at run time.

1. Introduction

Tracing just-in-time compilers (TJITs) determine frequently executed traces (hot paths and loops) at run time. These traces are then further analyzed and optimized, and finally specialized machine code is generated.

Traces contain guards that check whether a later execution will actually follow along the sequence of instructions originally recorded in the trace. For example, an if-statement in the program code gives rise to a guard, that will encode whether the recorded trace follows the then or the else branch. The guards are preserved in the specialized machine code generated for the optimized trace; when the guard condition does not hold later on, execution would transfer back from the optimized trace code to the original unoptimized code. If such a trace exit is taken frequently for a particular guard, a dedicated trace starting from that guard might be recorded. The result is a *trace tree* [4] where guards can have traces attached to it. The only control-flow join point is in the case of loops at the end of a trace that goes back to the loop head. Figure 1 illustrates the structure of trace trees, as they are implemented in our TJIT system SPUR [2].

Tracing at run time enables optimizations that cannot be performed ahead of time. For example, tracing performs inlining of method calls, including virtual calls, which give rise to a guard that ensures the virtual method table lookup yields the inlined method. As a result, traces spanning multiple inlined virtual method calls often contain redundant computations and guards. Eliminating this redundancy is the goal of trace analysis and optimizations.

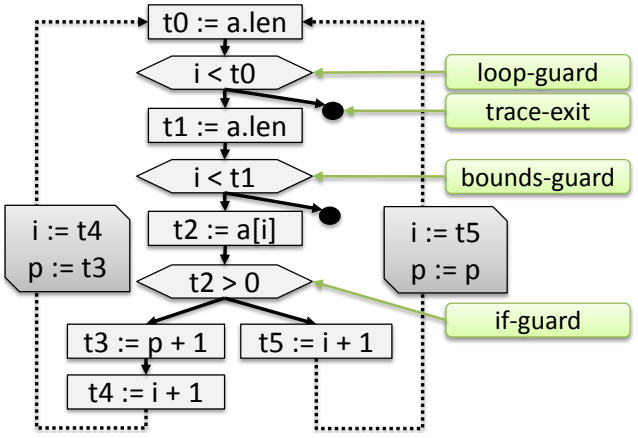

Figure 1. SPUR trace tree in SSA form with two traces for the loop **for** (i=0; i<a.len; ++i) { **if** (a[i]>0) p++; }

Up to now, TJITs employed standard compiler construction algorithms to analyze and optimize traces, often trying to balance the efficiency of the analysis with the efficiency of the resulting code, in order to achieve overall optimal performance. Consider a system where the unoptimized code quality is already reasonable, e.g., by always employing a baseline JIT instead of an interpreter, and where spare processing power is available on separate cores. Then, once traces have been recorded, a TJIT can perform expensive optimizations in parallel to the ongoing program execution.

We propose to leverage automated theorem provers to optimize traces at run time. In particular, Satisfiability Modulo Theories (SMT) solvers check satisfiability of formulas using decision procedures for theories such as equality, uninterpreted functions, arrays, and bitvector arithmetic. These theories are well suited to encode the semantics of the instructions of typical execution environments, and thus SMT solvers are often used in software and hardware verification, as well as in automatic test case generation.

SMT solvers based on DPLL(T) [5] architecture (i.e., all of them as of now) maintain a stack of asserted constraints. This stack is often exposed (e.g., in the recent SMT-LIB 2.0

standard [1]), so one can assert some constraints and pop them later. In between, one can check if a formula logically follows in the current context (by checking satisfiability of its negation). This perfectly matches the typical behavior of a trace tree analysis or optimization, which usually visits the trace tree in a depth-first manner.

2. Trace tree optimizations with SMT solver

The basic idea is to visit paths of a previously recorded trace tree in a depth-first manner, performing symbolic execution [6] along each path, pushing and popping constraints to the SMT solver to perform backtracking. Whenever an instruction is visited, its semantics are mapped to the corresponding theories of the SMT solver. For example, an instruction t3:=p+1 that adds one to a 32-bit p and stores the result in a local variable gives rise to an expression $\mathit{bvadd}(p, 1{:}\mathit{bv32})$ in the theory of bitvectors in the SMT solver; whenever the local variable $t3$ is used in the following, the expression $\mathit{bvadd}(p, 1{:}\mathit{bv32})$ is used in the SMT solver. The implicit heap is made explicit, e.g., an instruction t1 := a.len that reads field len of the object pointed to by a is modeled in the theory of arrays with the expression $\mathit{read}(h_{\mathsf{len}}, a)$ where h_{len} represents a mathematical map of object reference to values for the field len. An instruction that writes to a field of an object, e.g., x.f := y, gives rise to a new heap h'_{f} represented by $\mathit{write}(h_{\mathsf{f}}, x, y)$. Whenever a guard is visited, its condition is asserted as a fact to the SMT solver. When backtracking later to start the analysis of another trace, all previously asserted guards up to the backtracking point are popped off the stack of the solver.

Forward guard elimination. Before asserting the next guard condition, a solver query can be performed to check if the next guard condition follows from earlier asserted conditions. If so, the next guard can never fail, and can be removed.

Redundant-store elimination. For every write-operation (including to local variables), a solver query can be performed, checking whether the written value is guaranteed to be equal to the value that would be retrieved by a read-operation at the place of the write-operation. For example, for the write-operation x.f := y, the query would be $\mathit{read}(h_{\mathsf{f}}, x) = y$. If so, the write-operation can be removed.

Common-subexpression elimination modulo theories and asserted guards, including alias-analysis and redundant-load elimination. For every expression that is constructed in the course of symbolic execution, queries can be performed to check whether that expression is guaranteed to be equal to any expression which was constructed earlier along the same path. As an optimization, the model of the currently asserted constraints can be obtained. Only pairs of expressions with the same value in that model need to be considered. If an earlier equivalent expression is found, all references to the later expression can be replaced by the earlier equivalent expression, and the redundant later expression can be removed.

Speculative guard strengthening If a later guard is not implied by an earlier guard, it might still be the case that the later guard implies an earlier guard. This is not detected by the *forward guard elimination* described above. One approach to eliminate the implied guard in this scenario is to move the later guard before the earlier guard, so that forward guard elimination can remove it. Moving up a guard in this way is a speculative optimization, as it might cause execution to leave the trace earlier than it normally would. However, if done properly it does not change the semantics of the program, as it simply means that execution may leave the trace a bit early, and continue in unoptimized code. A guard can be moved up over the preceding instruction if the result or side effects of the preceding instruction do not affect the guard condition. By making side effects on the heap explicit this can be decided by a solver query.

3. Discussion and future work

The optimizations above rely only on the solver being sound, i.e., on the solver never claiming a satisfiable formula to be unsatisfiable. However, most SMT solvers are also complete for the fragment we need, and so is our encoding. Thus, the SMT solver is an ultimate oracle: if it determines that, e.g., a guard cannot be eliminated, then there exists an input on which the guard will fail. Thus, we employ the strongest possible version of this (conservative) optimization. In practice, we might want to limit the solver's running time, thus making it incomplete.

We are in the process of implementing trace tree optimizations such as the ones described above in our tracing JIT compiler SPUR [2] using Z3 [3] as the SMT solver, and expect to have results by the time of the conference.

References

[1] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0, 2010. Available at www.SMT-LIB.org.

[2] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A Trace-Based JIT Compiler for CIL. Technical Report MSR-TR-2010-27, Microsoft Research, 2010.

[3] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[4] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, University of California, Irvine, 2006.

[5] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *CAV'04*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[6] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.