SPEED: Symbolic Complexity Bound Analysis

Invited Talk

Sumit Gulwani

Microsoft Research, Redmond, USA
sumitg@microsoft.com

Abstract. The SPEED project addresses the problem of computing symbolic computational complexity bounds of procedures in terms of their inputs. We discuss some of the challenges that arise and present various orthogonal/complementary techniques recently developed in the SPEED project for addressing these challenges.

1 Introduction

As processor clock speeds begin to plateau, there is an increasing need to focus on software performance. One of the performance metrics is (worst-case) symbolic computational complexity bounds of procedures (expressed as a function of their inputs). Such automatically generated bounds are useful in early detection of egregious performance problems in large modular codebases that are constantly being changed by multiple developers who make heavy use of code written by others without a good understanding of their implementation complexity. These worst-case bounds also help augment the traditional performance measurement process of profiling, which is only as good as the set of test inputs.

The SPEED project develops static program analysis techniques for computing symbolic computational complexity bounds. Computing such bounds is a technically challenging problem since bounds for even simple sequential programs are usually disjunctive, non-linear, and involve numerical properties of heaps. Sometimes even proving termination is hard in practice, and computing bounds ought to be a harder problem.

This paper briefly describes some techniques that enable existing off-the-shelf linear invariant generation tools to compute non-linear and disjunctive bounds. These techniques include: (i) program transformation (control-flow refinement to transform loops with sophisticated control flow into simpler loops [7]), and (ii) monitor instrumentation (multiple counter variables whose placement is determined dynamically by the analysis [9]). This paper also briefly describes some specialized invariant generation tools (based on abstract interpretation) that enable bound computation. These can compute invariants that describe (i) numerical properties of memory partitions [8], (ii) relationships involving non-linear operators such as logarithm, exponentiation, multiplication, square-root, and `Max` [6]. These techniques together enable generation of complexity bounds that

are usually precise not only in terms of the computational complexity, but also in terms of the constant factors.

The hard part in computing computational complexity bound for a procedure is to compute bounds on the number of iterations of various loops inside that procedure. Given some cost measure for atomic statements, loop iteration bounds can be composed together in an easy manner to obtain procedure bounds (for details, see [9, 7].) This paper is thus focused on bounding loop iterations, and is organized by some of the challenges that arise in bounding loop iterations.

2 Loops with Control-flow

Loops with non-trivial control flow inside them often have (iteration) bounds that are non-linear or disjunctive, i.e., they involve use of the `Max` operator (which returns the maximum of its arguments). For example, the loop in Figure 1(a) has a disjunctive bound: $100 + \texttt{Max}(0, m)$, while the loop in Figure 2(a) has a non-linear bound: $n \times (m + 1)$. Such bounds can be computed using one of the following three techniques.

2.1 Single Counter Instrumentation

This technique involves instrumenting a counter i that is initialized to 0 at the beginning of the loop, and is incremented by 1 inside the loop. An invariant generation tool is then used to compute invariants that relate the loop counter i with program variables. Existential elimination of temporary variables (all variables except the counter variable i and the inputs) yields a relation between i and the inputs, from which an upper bound u on i may be read. $\texttt{Max}(0, u)$ then provides a bound on the number of loop iterations.

For the loop in Figure 1(a), single counter instrumentation results in the loop in Figure 1(b). Bound computation now requires computing the disjunctive inductive invariant $i \leq x + y + 1 \ \wedge \ y \leq \texttt{Max}(0, m) \ \wedge \ x < 100$ at program point 5 (i.e., at the location immediately before the statement at that point) in Figure 1(b). Existential elimination of temporary variables x and y from this inductive invariant yields the invariant $i \leq 100 + \texttt{Max}(0, m)$, which implies a bound of $\texttt{Max}(0, 100 + \texttt{Max}(0, m)) = 100 + \texttt{Max}(0, m)$ on the number of loop iterations.

For the loop in Figure 2(a), single counter instrumentation results in the loop in Figure 2(b). Bound computation now requires computing the non-linear inductive invariant $i \leq x \times m + x + y + 1 \ \wedge \ x < n \ \wedge \ y \leq m$ at program point 5 in Figure 2(b). Existential elimination of temporary variables x and y from this inductive invariant yields the invariant $i \leq n \times (m + 1)$, which implies a bound of $n \times (m + 1)$ on the number of loop iterations.

One (semi-automatic) technique to compute such disjunctive and non-linear invariants is to use the numerical abstract domain described in [6]. The numerical abstract domain is parametrized by a base linear arithmetic abstract domain and is constructed by means of two domain lifting operations that extend the

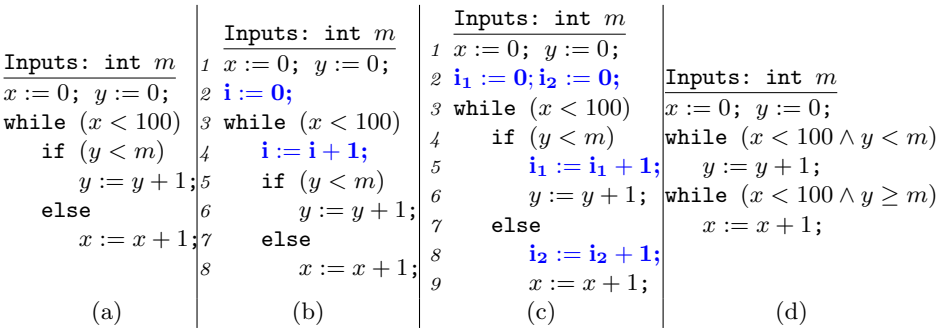

Fig. 1. (a) Loop with a disjunctive bound $100 + \texttt{Max}(0, m)$. (b) Single Counter Instrumentation requires computing disjunctive invariants for bound computation. (c) Multiple Counter Instrumentation enables computation of disjunctive bounds using linear invariants on individual counters. (d) Control-flow Refinement enables bound computation by reducing original loop to a code-fragment with simpler loops.

base linear arithmetic domain to reason about the max operator and other operators whose semantics is specified using a set of inference rules. One of the domain lifting operation extends the linear arithmetic domain to represent linear relationships over variables as well as max-expressions (an expression of the form $\texttt{Max}(e_1, \ldots, e_n)$ where e_i's are linear expressions). Another domain lifting operation lifts the abstract domain to represent constraints not only over program variables, but also over expressions from a given finite set of expressions S. The semantics of the operators (such as multiplication, logarithm, etc.) used in constructing expressions in S is specified as a set of inference rules. The abstract domain retains efficiency by treating these expressions just like any other variable, while relying on the inference rules to achieve precision.

2.2 Multiple Counter Instrumentation

This technique (described in [9]) allows for using a less sophisticated invariant generation tool at the cost of using a more sophisticated counter instrumentation scheme. In particular, this technique involves choosing a set of counter variables and for each counter variable selecting the locations to initialize it to 0 and the locations to increment it by 1. The counters and their placement are chosen such that (besides some completeness constraints) a given invariant generation tool can compute bounds on the counter variables at appropriate locations in terms of the procedure inputs. (There is a possibility that no such counter placement is possible, but if there is one, then the algorithm described in Section 4 in [9] will compute one.) The bounds on individual counter variables are then composed together appropriately to obtain a (potentially disjunctive or non-linear) bound on the total number of loop iterations (For details, see Theorem 1 in Section 3 in [9]).

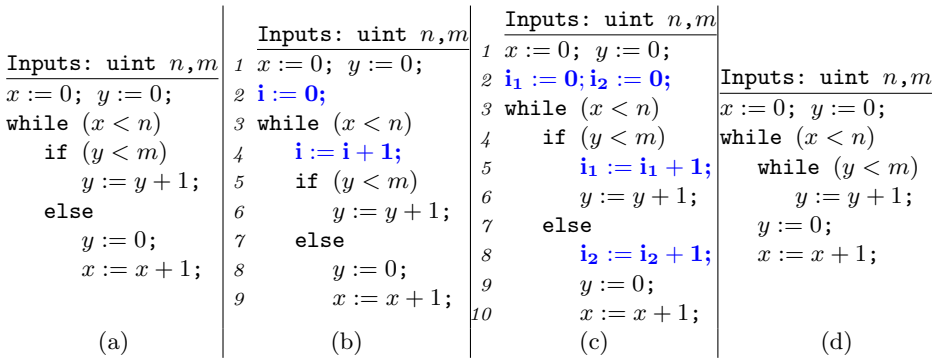

Fig. 2. `uint` denotes an unsigned (non-negative) integer. (a) Loop with a non-linear bound $n\times(m+1)$. (b) Single Counter Instrumentation requires computing non-linear invariants for bound computation. (c) Multiple Counter Instrumentation enables computation of non-linear bounds using linear invariants on individual counters. (d) Control-flow Refinement enables bound computation by reducing original multi-path loop to a code-fragment in which path-interleaving has been made more explicit.

An advantage of this technique is that in most cases (including the loops in Figure 1(a) and Figure 2(a)), it allows use of off-the-shelf linear invariant generation tools to compute disjunctive and non-linear bounds.

For the loop in Figure 1(a), the multiple counter instrumentation technique instruments the loop with two counters i_1 and i_2, both of which are initialized to 0 before the loop and are incremented by 1 on either sides of the conditional resulting in the loop shown in Figure 1(b). Now, consider the following (a bit subtle) argument with respect to computing bounds after this instrumentation. (All of this is automated by techniques described in [9].)

- If the then-branch is ever executed, it is executed for at most m iterations. This bound m can be obtained by computing a bound on counter variable i_1 at program point 6, which has been instrumented to count the number of iterations of the then-branch.
- Similarly, if the else-branch is ever executed, it is executed for at most 100 iterations. This bound can be obtained by computing a bound on counter variable i_2 at program point 9, which has been instrumented to count the number of iterations of the else-branch.
- This implies that the total number of loop iterations is bounded by $\texttt{Max}(0, m)+\texttt{Max}(0, 100) = 100 + \texttt{Max}(0, m)$.

For the loop in Figure 2(a), the multiple counter instrumentation technique instruments the loop with two counters i_1 and i_2 that are initialized and incremented as shown in Figure 2(c). Now, consider the following argument with respect to computing bounds after this instrumentation.

- The else-branch is executed for at most n iterations. This bound n can be obtained by computing a bound on counter variable i_2 at program point 9,

which has been instrumented to count the number of iterations of the else-branch.
- The number of iterations of the then-branch in between any two iterations of the else-branch is bounded by m. This bound can be obtained by computing a bound on counter variable i_1 at program point 6, which is incremented by 1 in the then-branch, but is re-initialized to 0 in the else-branch.
- This implies that the total number of loop iterations is bounded by $\texttt{Max}(0, n) \times (1 + \texttt{Max}(0, m)) = n \times (m + 1)$.

2.3 Control-Flow Refinement

This technique (described in [7]) allows for using a less sophisticated invariant generation tool, but in a two phase process. The first phase consists of performing *control-flow refinement*, which is a semantics-preserving and bound-preserving transformation on procedures. Specifically, a loop that consists of multiple paths (arising from conditionals) is transformed into a code fragment with one or more simpler loops in which the interleaving of paths is syntactically explicit. For the loops in Figure 1(a) and in Figure 2(a), the control-flow refinement leads to the code-fragments in Figure 1(d) and Figure 2(d) respectively.

The second phase simply consists of performing the analysis on the refined procedure. The code-fragment with transformed loops enables a more precise analysis than would have been possible with the original loop. The additional program points created by refinement allow the invariant generator to store more information. The invariants at related program points (which map to the same program point in the original loop) in the refined code-fragment correspond to a disjunctive invariant at the corresponding program point in the original loop. For the loops in Figure 1(d) and Figure 2(d), the desired bounds can now be easily computed using simple techniques for bound computation such as pattern matching, or single counter instrumentation.

Even though the goals of the multiple counter instrumentation and control-flow refinement techniques are the same (that of enabling bound computation with a less sophisticated invariant generation tool), there are examples for which one of them is preferable than the other. (See the motivating examples in the respective papers [9, 7].) An interesting strategy that leverages the power of both techniques would be to use multiple counter instrumentation technique after applying control-flow refinement.

3 Loops in a Nested Context

We now consider the case of a loop nested inside another loop. If our goal is to only prove termination of the outer loop, we can perform the process in a modular fashion: prove that the number of iterations of the outer loop is bounded, and prove that the number of iterations of the inner loop, in between any two iterations of the outer loop, is bounded. However, such a modular scheme would not work well for bound computation since it may often yield conservative

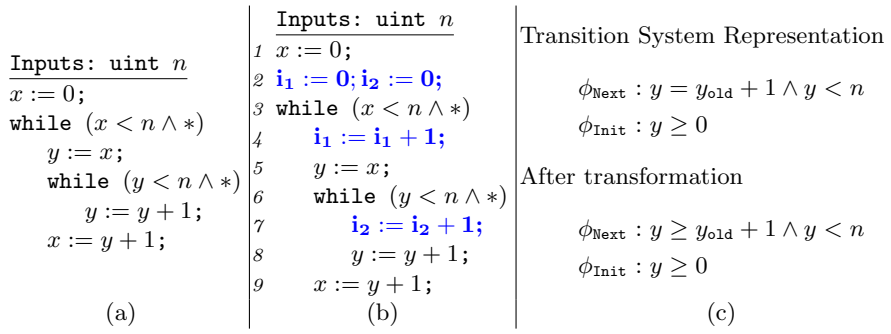

Fig. 3. $*$ denotes non-deterministic choice. (a) Nested Loop with a bound of n on the total number of iterations. (b) Counter Instrumentation methodology can be used to compute precise bounds for nested loops by instrumenting counters in an appropriate manner. (c) Transition system of the nested loop in the context of the outer loop, as well as when lifted out of the context of its outer loop. The latter can be used to obtain a precise bound on the total number of loop iterations of the nested loop.

bounds, as is illustrated by the example in Figure 3(a). The bound for the inner loop (in terms of the inputs), for each iteration of the outer loop, is n. However, the interesting part to note is that the bound for the total number of iterations of the inner loop is also n. It is quite often the case that the nesting depth of a procedure is not indicative of the precise computational complexity.

Both the counter instrumentation and control-flow refinement techniques introduced in Section 2 can be extended to deal with the challenge of computing precise bounds for loops in a nested context.

Appropriate Counter Instrumentation

The single counter instrumentation methodology can be easily extended to nested loops by using a counter that is incremented by 1 in the nested loop (as before), but is initialized to 0 at the beginning of the procedure (as opposed to immediately before the loop). The idea of multiple counter instrumentation can be similarly extended to address nested loops by having appropriate constraints on the positioning of multiple counters. (For more details, see the notion of a *proof structure* in Section 3 in [9].) In either case, the loop in Figure 3(a) is instrumented as shown in Figure 3(b). The bound of n on the total number of iterations of the nested loop now follows from the inductive invariant $i_2 \leq y \ \wedge \ y < n$ at program point 8 in Figure 3(b).

Appropriate Loop Re-structuring

Alternatively, the nested loop can be re-structured as a non-nested loop in a bound-preserving transformation. This is easier illustrated by working with a

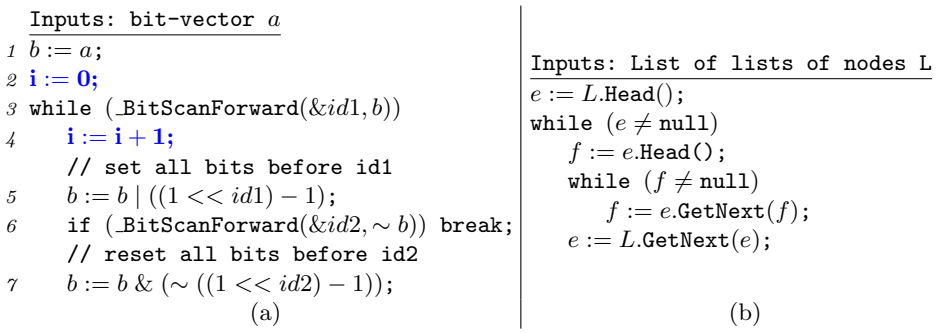

Fig. 4. Loops whose bound cannot be expressed using scalar program variables, and instead require reference to quantitative functions of the data-structures over which they iterate. (a) Loop (instrumented with a counter variable i) that iterates over bit-vectors. `_BitScanForward` returns the bit position of the first set bit in the first parameter (if any, which is signaled by the return value) (b) Loop that iterates over a list of lists.

transition system representation of the loop. A transition system can be described by a relation $\phi_{\texttt{Next}}$ representing relations between loop variables y and their values $y_{\texttt{old}}$ in the previous iteration, and a relation $\phi_{\texttt{Init}}$ representing the initial value of the loop variables.

For the nested loop in Figure 3(a), the transition system representation before and after this transformation is illustrated in Figure 3(c). The interesting part to note is that the relation $y = y_{\texttt{old}} + 1$ is transformed into $y \geq y_{\texttt{old}} + 1$ when we compare any two successive states inside the nested loop, but not necessarily inside the same iteration of the outer loop. (Observe that the value of y increases monotonically inside the nested loop.) The desired precise bound of n for the nested loop can now be easily computed from the transformed transition system (by using simple techniques for bound computation such as pattern matching, or single counter instrumentation.)

The above-mentioned transformation of the transition system can be carried out by making use of *progress invariants* as described in Section 5 in [7].

4 Loops iterating over Data-structures

We now consider loops that iterate over data-structures such as lists, trees, bit-vectors, etc. In such cases, it may not be possible to express bounds using only scalar program variables. However, bounds may be expressible in terms of some quantitative functions of these data-structures (such as length of a list, or height of a tree, or number of bits in a bit-vector).

[9] proposes the notion of user-defined quantitative functions, wherein the user specifies the semantics of the quantitative functions by annotating each method of an abstract data-structure with how it may affect the quantitative

attributes of the input data-structures, and how it determines the quantitative attributes of the output data-structures. Bounds can then be obtained by computing invariants that relate the instrumented loop counter with the quantitative functions of data-structures. Such invariants can be obtained by using a numerical invariant generation tool that has been extended with support for uninterpreted functions [10] and aliasing.

The loop in Figure 4(a) iterates over a bit-vector by masking out the least significant consecutive chunk of 1s from b in each loop iteration. Bit-vectors have quite a few interesting quantitative functions associated with them. E.g., $\texttt{Bits}(a)$: total number of bits, $\texttt{Ones}(a)$: total number of 1 bits, $\texttt{One}(a)$: position of the least significant 1 bit, etc. Using these quantitative functions, it is possible to express relationships between the instrumented loop counter i and the bit-vectors a and b. In particular, it can be established that $2i \leq 1 + \texttt{One}(b) - \texttt{One}(a) \ \wedge \ i \leq 1 + \texttt{Ones}(a) - \texttt{Ones}(b)$ at program point 5. This implies bounds of both $\texttt{Ones}(a)$ as well as $(\texttt{Bits}(a) - \texttt{One}(a))/2$ on the number of loop iterations.

The loop in Figure 4(b) iterates over a list of lists of nodes. The outer loop iterates over the top-level list, while the inner loop processes all nodes in the nested lists. The iterations of the outer loop are bounded above by the length of the top-level list, while the total iterations of the nested loop are bounded above by the sum of the lengths of all the nested lists. These bounds can be expressed using appropriate quantitative functions $\texttt{Len}(L)$ and $\texttt{TotalNodes}(L)$ with the expected semantics. However, computation of these bounds requires inductive invariants that require a few more quantitative functions. (For more details, see Section 5.2 in [9].)

5 Loops with Complex Progress Measure

There are loops with simple control flow (and hence simplification techniques like multiple counter instrumentation or control-flow refinement do not help reduce the complexity of bound computation) that require computing sophisticated invariants for establishing bounds. We discuss below two such classes of invariants.

Invariants that relate sizes of memory partitions

Consider the BubbleSort procedure shown in Figure 5(a) that sorts an input array A of length n. The algorithm works by repeatedly iterating through the array to be sorted, comparing two items at a time and swapping them if they are in the wrong order (Line 8). The iteration through the array (Loop in lines 6-9) is repeated until no swaps are needed (measured by the `change` boolean variable), which indicates that the array is sorted.

Notice that establishing a bound on the number of iterations of the outer while-loop of this procedure is non-trivial; it is not immediately clear why the outer while-loop even terminates. Note that in each iteration of the while-loop, at least one new element "bubbles" up to its correct position in the array (i.e.,

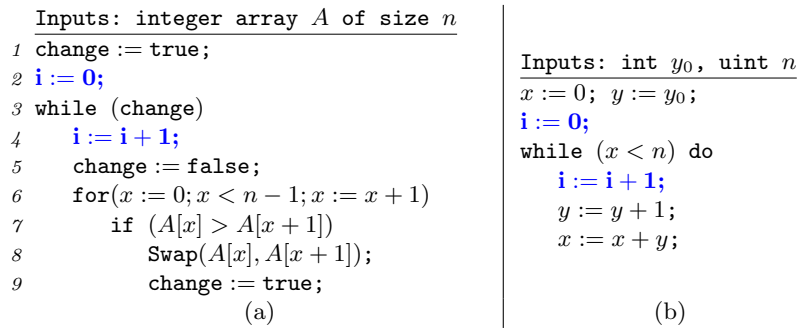

Fig. 5. Loops where multiple counter instrumentation and control-flow refinement do not help reduce the complexity of bound computation, and instead sophisticated invariants are required to relate the instrumented counter variable i with program variables. (a) BubbleSort procedure (instrumented with a counter variable i to bound the number of iterations of the outer loop) whose bound computation requires computing numeric invariants over sizes of appropriate memory partitions. (b) Loop (instrumented with a counter variable i) whose bound computation requires computing invariants with non-linear operators, in particular, multiplication and square-root.

it is less than or equal to all of its successors). Hence, the outer while-loop terminates in at most n steps. The set cardinality abstract domain described in [8] can be used to automatically establish this invariant by computing a relationship between the instrumented loop counter i and the number of elements that have been put in the correct position. In particular, the set cardinality analysis computes the invariant that i is less than or equal to the size of the set of the array indices that hold elements at their correct position (provided the parametrized set cardinality analysis is constructed from a set analysis whose base-set constructor can represent such a set).

Invariants with non-linear operators

Consider the loop shown in Figure 5(b) (taken from [3], which uses the principle of second-order differences to establish a lexicographic polyranking function for proving termination). This loop illustrates the importance of using non-linear operators like multiplication and square-root for representing timing bounds as well as computing the invariants required to establish timing bounds. In particular, it is possible to compute a bound of $\sqrt{2n} + \texttt{max}(0, -2y_0) + 1$ on the counter variable i after establishing the inductive loop invariant $i = y - y_0 \wedge y^2 \leq y_0^2 + 2x$. Such invariants can be computed by the numerical abstract domain [6] briefly described in Section 2.1.

6 Related Work

WCET Analysis: There is a large body of work on estimating worst case execution time (WCET) in the embedded and real-time systems community [20, 21]. The WCET research is largely orthogonal, focused on distinguishing between the complexity of different code-paths and low-level modeling of architectural features such as caches, branch prediction, instruction pipelines. For establishing loop bounds, WCET techniques either require user annotation, or use simple techniques based on pattern matching [13] or some relatively simple numerical analysis (e.g., relational linear analysis to compute linear bounds on the delay or timer variables of the system [12], interval analysis based approach [11], symbolic computation of integer points in a polyhedra [14]). These WCET techniques cannot compute precise bounds for several examples considered in this paper.

Termination Techniques: Recently, there have been some new advances in the area of proving termination of loops based on discovering disjunctively well-founded ranking functions [17] or lexicographic polyranking functions [4]. [5, 2] have successfully applied the fundamental result of [17] on disjunctively well-founded relations to prove termination of loops in real systems code. It is possible to obtain bounds from certain kind of ranking functions given the initial state at the start of the loop. However, the ranking function abstraction is sometimes too weak to compute precise bounds. In contrast, computation of any (finite) bound for a loop is a proof of its termination.

Symbolic Bound Computation using Recurrence Solving: A common approach to bound analysis has been that of generating recursive equations from programs such that a closed form solution to the recursive equations would provide the desired bounds. The process of generating recursive equations is fairly standard and syntactic; the challenging part lies in finding closed-form solutions. Various techniques have been proposed for finding closed-form solutions such as rewrite rules [15, 18], building specialized recurrence solvers using standard mathematical techniques [19], or using existing computer algebra systems. Recently, [1] have proposed a new technique based on computing ranking functions, loop invariants, and partial evaluation to more successfully solve recurrence relations that arise in practice. This recurrence relation approach does not directly address the challenges of dealing with loops with complicated progress measure or loops that iterate over data-structures. It would however be interesting to compare this approach with the techniques mentioned in this paper over numerical loops with non-trivial control-flow and those that occur in a nested context.

7 Conclusion and Future Work

Computing symbolic complexity bounds is a challenging problem, and we have applied a wide variety of static analysis techniques to address some of the involved challenges: new invariant generation tools (for computing invariants that

are disjunctive, non-linear, and can express numerical properties of heap partitions), monitor instrumentation (multiple counter instrumentation), program transformation (control-flow refinement), and user annotations (user-defined quantitative functions of data-structures).

The techniques described in this article are applicable for computing bounds of sequential procedures. Computing bounds of procedures in a concurrent setting is a more challenging problem: it requires modeling the scheduler, and bounds would be a function of the number of processors.

It would also be interesting to extend these techniques to compute bounds on other kind of resources (besides time) used by a program, such as memory or network bandwidth, or some user-definable resource [16]. Computing memory bounds is more challenging since unlike time its consumption does not monotonically increase with program execution because of de-allocation.

Acknowledgments The SPEED project has received contributions from several colleagues: Trishul Chilimbi, Bhargav Gulavani, Sagar Jain, Eric Koskinen, Tal Lev-Ami, Krishna Mehra, Mooly Sagiv. We thank the product teams at Microsoft for their assistance with this project. We thank Varun Aggarwala and Aditya Nori for providing useful feedback on a draft of this paper.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*, pages 221–237, 2008.
2. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.
3. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
4. A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, volume 3580 of *LNCS*, pages 1349–1361, 2005.
5. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.
6. B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *CAV*, pages 370–384, 2008.
7. S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, 2009.
8. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.
9. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.
10. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.
11. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, pages 57–66, 2006.
12. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *FMSD*, 11(2), 1997.

13. C. A. Healy, M. Sjodin, V. Rustagi, D. B. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.
14. B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET*, 2003.
15. D. L. Métayer. Ace: An Automatic Complexity Evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.
16. J. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP*, pages 348–363, 2007.
17. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE, July 2004.
18. M. Rosendahl. Automatic Complexity Analysis. In *FPCA*, pages 144–156, 1989.
19. B. Wegbreit. Mechanical Program Analysis. *Commun. ACM*, 18(9):528–539, 1975.
20. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrm. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. In *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.
21. R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. In *CAV*, pages 22–36, 2008.