Compositional Certified Resource Bounds

Quentin Carbonneaux Jan Hoffmann Zhong Shao
Yale University, USA
{quentin.carbonneaux, jan.hoffmann, zhong.shao}@yale.edu

Abstract

This paper presents a new approach for automatically deriving worst-case resource bounds for C programs. The described technique combines ideas from amortized analysis and abstract interpretation in a unified framework to address four challenges for state-of-the-art techniques: compositionality, user interaction, generation of proof certificates, and scalability. *Compositionality* is achieved by incorporating the potential method of amortized analysis. It enables the derivation of global whole-program bounds with local derivation rules by naturally tracking size changes of variables in sequenced loops and function calls. The resource consumption of functions is described abstractly and a function call can be analyzed without access to the function body. *User interaction* is supported with a new mechanism that clearly separates qualitative and quantitative verification. A user can guide the analysis to derive complex non-linear bounds by using auxiliary variables and assertions. The assertions are separately proved using established qualitative techniques such as abstract interpretation or Hoare logic. *Proof certificates* are automatically generated from the local derivation rules. A soundness proof of the derivation system with respect to a formal cost semantics guarantees the validity of the certificates. *Scalability* is attained by an efficient reduction of bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. The analysis framework is implemented in the publicly-available tool C^4B. An experimental evaluation demonstrates the advantages of the new technique with a comparison of C^4B with existing tools on challenging micro benchmarks and the analysis of more than 2900 lines of C code from the cBench benchmark suite.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability

Keywords Quantitative Verification, Resource Bound Analysis, Static Analysis, Amortized Analysis, LP Solving, Program Logic

PLDI'15, June 13–17, 2015, Portland, OR, USA.

ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2737955

1. Introduction

In software engineering and software verification, we often would like to have static information about the quantitative behavior of programs. For example, stack and heap-space bounds are important to ensure the reliability of safety-critical systems [30]. Static energy usage information is critical for autonomous systems and has applications in cloud computing [15, 16]. Worst-case time bounds can help create constant-time implementations that prevent side-channel attacks [7, 28]. Loop and recursion-depth bounds are used to ensure the accuracy of programs that are executed on unreliable hardware [12] and complexity bounds are needed to verify cryptographic protocols [6]. In general, quantitative resource information can provide useful feedback for developers.

Available techniques for *automatically* deriving worst-case resource bounds fall into two categories. Techniques in the first category derive impressive bounds for numerical imperative programs, but are not compositional. This is problematic if one needs to derive global whole-program bounds. Techniques in the second category derive tight whole-program bounds for programs with regular loop or recursion patterns that decrease the size of an individual variable or data structure. They are highly compositional, scale for large programs, and work directly on the syntax. However, they do not support multivariate interval-based resource bounds (e.g., $x - y$) which are common in C programs. Indeed, it has been a long-time open problem to develop compositional resource analysis techniques that can work for typical imperative code with non-regular iteration patterns, signed integers, mutation, and non-linear control flow.

Tools in the first category include SPEED [20], KoAT [11], PUBS [1], Rank [3], and LOOPUS [31]. They lack compositionality in at least two ways. First, they all base their analysis on some form of *ranking function* or *counter instrumentation* that is linked to a local analysis. As a result, loop bounds are arithmetic expressions that depend on the values of variables just before the loop. This makes it hard to give a resource bound on a sequence of loops and function calls in terms of the input parameters of a function. Second, while all popular imperative programming languages provide a function or procedure abstraction, available tools are not able to abstract resource behavior; instead, they have to inline the procedure body to perform their analysis.

Tools in the second category originate form the *potential method* of amortized analysis and type systems for functional programs [24, 25]. It has been shown that class definitions of object-oriented programs [26] and data-structure predicates of separation logic [5] can play the role of the type system in imperative programs. However, a major weakness of existing potential-based techniques is that they can only associate *potential* with individual program variables or data structures. For C programs, this fails for loops as simple as `for(i=x;i<y;i++)` where $y - i$ decreases, but not $|i|$.

A general problem with existing tools (in both categories) is user interaction. When a tool fails to find a resource bound for a program, there is no possibility for sound user interaction to guide

the tool during bound derivation. For example, there is no concept of manual proofs of resource bounds; and no framework can support composition of manually derived bounds with automatically inferred bounds.

This paper presents a new compositional framework for automatically deriving resource bounds on C programs. This new approach is an attempt to unify the two aforementioned categories: It solves the compositionality issues of techniques for numerical imperative code by adapting amortized-analysis–based techniques from the functional world. Our automated analysis is able to infer resource bounds on C programs with mutually-recursive functions and integer loops. The resource behavior of functions can be summarized in a functional specification that can be used at every call site without accessing the function body. To our knowledge this is the first technique based on amortized analysis that is able to derive bounds that depend on negative numbers and differences of variables. It is also the first resource analysis technique for C that deals naturally with recursive functions and sequenced loops, and can handle resources that may become available during execution (e.g., when freeing memory). Compared to more classical approaches based on ranking functions, our tool inherits the benefits of amortized reasoning. Using only one simple mechanism, it handles:

- interactions between *sequential loops or function calls* through size changes of variables,
- *nested loops* that influence each other with the same set of modified variables,
- and *amortized bounds* as found, for example, in the Knuth-Morris-Pratt algorithm for string search.

The main innovations that make amortized analysis work on imperative languages are to base the analysis on a Hoare-like logic and to track multivariate quantities instead of program variables. This leads to precise bounds expressed as functions of sizes $|[x, y]| = \max(0, y - x)$ of intervals. A distinctive feature of our analysis system is that it reduces linear bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. This enables the efficient inference of global bounds for larger programs. Moreover, our local inference rules automatically generate proof certificates that can be easily checked in linear time.

The use of the potential method of amortized analysis makes user interaction possible in different ways. For one thing, we can directly combine the new automatic analysis with manually derived bounds in a previously-developed quantitative Hoare logic [13] (see Section 7). For another thing, we describe a new mechanism that allows the separation of quantitative and qualitative verification (see Section 6). Using this mechanism, the user can guide the analysis by using auxiliary variables and logical assertions that can be verified by existing qualitative tools such as Hoare logic or abstract interpretation. In this way, we can benefit from existing automation techniques and provide a middle-ground between fully automatic and fully manual verification for bound derivation. This enables the semi-automatic inference of non-linear bounds, such as polynomial, logarithmic, and exponential bounds.

We have implemented the analysis system in the tool C^4B and experimentally evaluated its effectiveness by analyzing system code and examples from the literature. C^4B has automatically derived global resource bounds for more than 2900 lines of C code from the cBench benchmark suite. The extended version of this article [14] contains more than 30 challenging loop and recursion patterns that we collected from open source software and the literature. Our analysis can find asymptotically tight bounds for all but one of these patterns, and in most cases the derived constant factors are tight. To compare C^4B with existing techniques, we tested our examples with tools such as KoAT [11], Rank [3], and LOOPUS [31]. Our experiments show that the bounds that we derive are often more precise than those derived by existing tools. Only LOOPUS [31], which also uses amortization techniques, is able to achieve a similar precision.

Examples from cBench and micro benchmarks demonstrate the practicality and expressiveness of the user guided bound inference. For example, we derive a logarithmic bound for a binary search function and a bound that amortizes the cost of k increments to a binary counter (see Section 6).

In summary, we make the following *contributions*.

- We develop the first automatic amortized analysis for C programs. It is naturally compositional, tracks size changes of variables to derive global bounds, can handle mutually-recursive functions, generates resource abstractions for functions, derives proof certificates, and handles resources that may become available during execution.
- We show how to automatically reduce the inference of *linear* resource bounds to efficient LP solving.
- We describe a new method of harnessing existing qualitative verification techniques to guide the automatic amortized analysis to derive *non-linear* resource bounds with LP solving.
- We prove the soundness of the analysis with respect to a parametric cost semantics for C programs. The cost model can be further customized with function calls ($\mathsf{tick}(\mathsf{n})$) that indicate resource usage.
- We implemented our resource bound analysis in the publicly-available tool C^4B.
- We present experiments with C^4B on more than 2900 lines of C code. A detailed comparison shows that our prototype is the only tool that can derive global bounds for larger C programs while being as powerful as existing tools when deriving linear local bounds for tricky loop and recursion patterns.

2. The Potential Method

The idea that underlies the design of our framework is amortized analysis [32]. Assume that a program S executes on a starting state σ and consumes n resource units of some user-defined quantity. We denote that by writing $(S, \sigma) \Downarrow_n \sigma'$ where σ' is the program state after the execution. The basic idea of amortized analysis is to define a *potential function* Φ that maps program states to non-negative numbers and to show that $\Phi(\sigma) \geqslant n$ if σ is a program state such that $(S, \sigma) \Downarrow_n \sigma'$. Then $\Phi(\sigma)$ is a valid resource bound.

To obtain a compositional reasoning we also have to take into account the state resulting from a program's execution. We thus use two potential functions, one that applies before the execution, and one that applies after. The two functions must respect the relation $\Phi(\sigma) \geqslant n + \Phi'(\sigma')$ for all states σ and σ' such that $(S, \sigma) \Downarrow_n \sigma'$. Intuitively, $\Phi(\sigma)$ must provide enough *potential* for both, paying for the resource cost of the computation and paying for the potential $\Phi'(\sigma')$ on the resulting state σ'. That way, if $(\sigma, S_1) \Downarrow_n \sigma'$ and $(\sigma', S_2) \Downarrow_m \sigma''$, we get $\Phi(\sigma) \geqslant n + \Phi'(\sigma')$ and $\Phi'(\sigma') \geqslant m + \Phi''(\sigma'')$. This can be composed as $\Phi(\sigma) \geqslant (n+m) + \Phi''(\sigma'')$. Note that the initial potential function Φ provides an upper bound on the resource consumption of the whole program. What we have observed is that, if we define $\{\Phi\}\, S\, \{\Phi'\}$ to mean

$$\forall \sigma\, n\, \sigma'.\, (\sigma, S) \Downarrow_n \sigma' \implies \Phi(\sigma) \geqslant n + \Phi'(\sigma')\ ,$$

then we get the following familiar looking rule

$$\frac{\{\Phi\}\, S_1\, \{\Phi'\} \qquad \{\Phi'\}\, S_2\, \{\Phi''\}}{\{\Phi\}\, S_1; S_2\, \{\Phi''\}}\ .$$

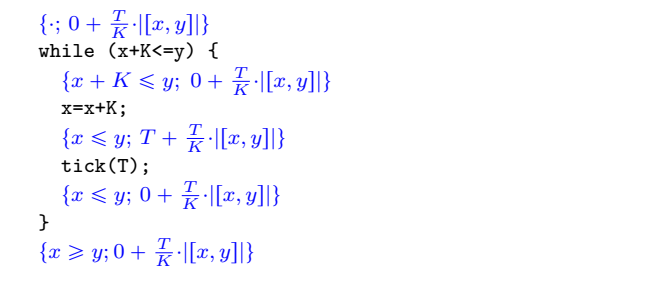


Figure 1. Derivation of a tight bound on the number of ticks for a standard *for loop*. The parameters $K > 0$ and $T > 0$ are not program variables but denote concrete constants.

This rule already shows a departure from classical techniques that are based on ranking functions. Reasoning with two potential functions promotes compositional reasoning by focusing on the sequencing of programs. In the previous rule, Φ gives a bound for $S_1; S_2$ through the intermediate potential Φ', even though it was derived on S_1 only. Similarly, other language constructs lead to rules for the potential functions that look very similar to Hoare logic or effect system rules. These rules enable reasoning about resource usage in a flexible and compositional way, which, as a side effect, produces a certificate for the derived resource bound.

The derivation of a resource bound using potential functions is best explained by example. If we use the tick metric that assigns cost n to the function call $\mathsf{tick(n)}$ and cost 0 to all other operations then the cost of the following example can be bounded by $|[x, y]| = \max(y-x, 0)$.

```
while (x<y) { x=x+1; tick(1); }
```
(Example 1)

To derive this bound, we start with the initial potential $\Phi_0 = |[x, y]|$, which we also use as the loop invariant. For the loop body we have (like in Hoare logic) to derive a triple $\{\Phi_0\}\, \mathsf{x = x + 1; tick(1)}\, \{\Phi_0\}$. We can only do so if we utilize the fact that $x < y$ at the beginning of the loop body. The reasoning then works as follows. We start with the potential $|[x, y]|$ and the fact that $|[x, y]| > 0$ before the assignment. If we denote the updated version of x after the assignment by x' then the relation $|[x, y]| = |[x', y]| + 1$ between the potential before and after the assignment $\mathsf{x = x + 1}$ holds. This means that we have the potential $|[x, y]| + 1$ before the statement $\mathsf{tick(1)}$. Since $\mathsf{tick(1)}$ consumes one resource unit, we end up with potential $|[x, y]|$ after the loop body and have established the loop invariant again.

Figure 1 shows a derivation of the bound $\frac{T}{K}\cdot|[x, y]|$ on the number of ticks for a generalized version of Example 1 in which we increment x by a constant $K > 0$ and consume $T > 0$ resources in each iteration. The reasoning is similar to the one of Example 1 except that we obtain the potential $K \cdot \frac{T}{K}$ after the assignment. In the figure, we separate logical assertions from potential functions with semicolons. Note that the logical assertions are only used in the rule for the assignment $\mathsf{x = x + K}$.

To the best of our knowledge, no other implemented tool for C is currently capable of deriving a tight bound on the cost of such a loop. For $T = 1$ (many systems focus on the number of loop iterations without a cost model) and $K = 10$, KoAT computes the bound $|x| + |y| + 10$, Rank computes the bound $y - x - 7$, and LOOPUS computes the bound $y-x-9$. Only PUBS computes the tight bound $0.1(y - x)$ if we translate the program into a term-rewriting system by hand. We will show in the following sections that the potential method makes automatic bound derivation straightforward.

The concept of a potential function is a generalization of the concept of a ranking function. A potential function can be used like a ranking function if we use the tick metric and add the statement $\mathsf{tick(1)}$ to every back edge of the program (loops and function calls). However, a potential function is more flexible. For example, we can use a potential function to prove that Example 2 does not consume any resources in the tick metric.

```
while (x<y) {tick(-1); x=x+1; tick(1)}
```
(Example 2)

```
while (x<y) { x=x+1; tick(10); }
```
(Example 3)

Similarly we can prove that Example 3 can be bounded by $10|[x, y]|$. In both cases, we reason exactly like in the first version of the while loop to prove the bound. Of course, such loops with different tick annotations can be seamlessly combined in a larger program.

3. Compositional Resource-Bound Analysis

In this section we describe the high-level design of the automatic amortized analysis that we implemented in C^4B. Examples explain and motivate our design decisions.

Linear Potential Functions. To find resource bounds automatically, we first need to restrict our search space. In this work, we focus on the following form of potential functions, which can express tight bounds for many typical programs and allows for inference with *linear programming*.

$$\Phi(\sigma) = q_0 + \sum_{x,y \in \mathrm{dom}(\sigma) \,\wedge\, x \neq y} q_{(x,y)} \cdot |[\sigma(x), \sigma(y)]|\,.$$

Here $\sigma : (\mathrm{Locals} \to \mathbb{Z}) \times (\mathrm{Globals} \to \mathbb{Z})$ is a simplified program state that maps variable names to integers, $|[a, b]| = \max(0, b - a)$, and $q_i \in \mathbb{Q}_0^+$. To simplify the references to the linear coefficients q_i, we introduce an *index set* I. This set is defined to be $\{0\} \cup \{(x, y) \mid x, y \in \mathit{Var} \wedge x \neq y\}$. Each index i corresponds to a *base function* f_i in the potential function: 0 corresponds to the constant function $\sigma \mapsto 1$, and (x, y) corresponds to $\sigma \mapsto |[\sigma(x), \sigma(y)]|$. Using these notations we can rewrite the above equality as $\Phi(\sigma) = \sum_{i \in I} q_i f_i(\sigma)$.We often write xy to denote the index (x, y). This allows us to uniquely represent any linear potential function Φ as a *quantitative annotation* $Q = (q_i)_{i \in I}$, that is, a family of non-negative rational numbers where only a finite number of elements are not zero.

In the potential functions, we treat constants as global variables that cannot be assigned to. For example, if the program contains the constant 1988 then we have a variable c_{1988} and $\sigma(c_{1988}) = 1988$. We assume that every program state includes the constant c_0.

Abstract Program State. In addition to the quantitative annotations, our automatic amortized analysis needs to maintain a minimal abstract state to justify certain operations on quantitative annotations. For example when analyzing the code $x \leftarrow x + y$, it is helpful to know the sign of y to determine which intervals will increase or decrease. The knowledge needed by our rules can be inferred by local reasoning (i.e. in basic blocks without recursion and loops) within usual theories (e.g. Presburger arithmetic or bit vectors).

The abstract program state is represented as *logical contexts* in the derivation system used by our automated tool. Our implementation finds these logical contexts using abstract interpretation with the domain of linear inequalities. We observed that the rules of the analysis often require only minimal local knowledge. This means that it is not necessary for us to compute precise loop invariants and only a rough fixpoint (e.g. keeping only inequalities on variables unchanged by the loop) is sufficient to obtain good bounds.

Challenging Loops. One might think that our set of potential functions is too simplistic to be able to express and prove bounds for realistic programs. Nevertheless, we can handle challenging example programs without special tricks or techniques. Examples

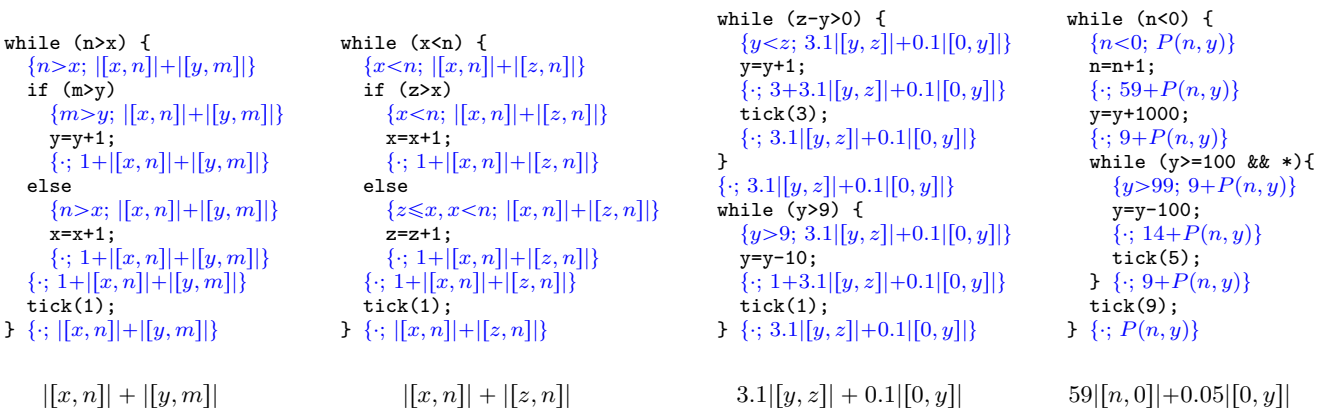


Figure 2. Derivations of bounds on the number of ticks for challenging examples. Examples *speed_1* and *speed_2* (from [20]) use *tricky iteration patterns*, *t08a* contains *sequential loops* so that the iterations of the second loop depend on the first, and *t27* contains interacting *nested loops*. In Example *t27*, we use the abbreviation $P(n,y) := 59|[n,0]|+0.05|[0,y]|$.

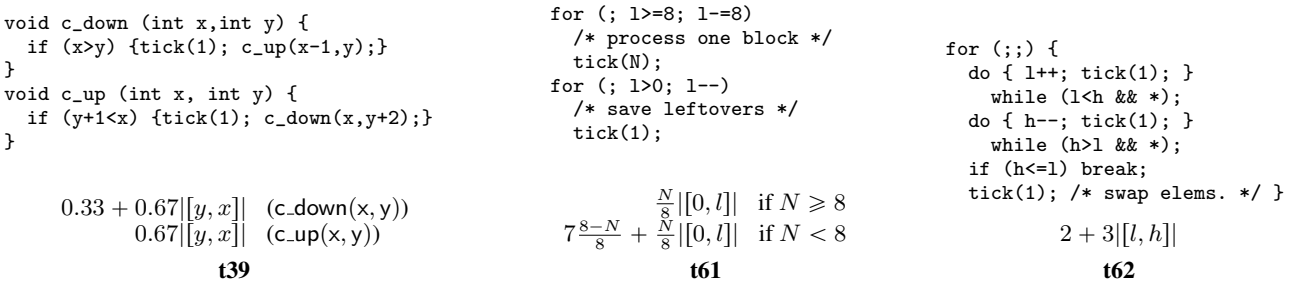


Figure 3. Example *t39* shows two mutually-recursive functions with the computed tick bounds. Example *t61* and *t62* demonstrate the unique compositionality of our system. In *t61*, $N \geqslant 0$ is a fixed but arbitrary constant.

speed_1 and *speed_2* in Figure 2, which are taken from previous work [20], demonstrate that our method can handle *tricky iteration patterns*. The SPEED tool [20] derives the same bounds as our analysis but requires heuristics for its counter instrumentation. These loops can also be handled with inference of *disjunctive invariants*, but in the abstract interpretation community, these invariants are known to be notoriously difficult to generate. In Example *speed_1* we have one loop that first increments variable y up to m and then increments variable x up to n. We derive the tight bound $|[x,n]| + |[y,m]|$. Example *speed_2* is even trickier, and we found it hard to find a bound manually. However, using potential transfer reasoning as in amortized analysis, it is easy to prove the tight bound $|[x,n]| + |[z,n]|$.

Nested and Sequenced Loops. Example *t08a* in Figure 2 shows the ability of the analysis to discover interaction between *sequenced loops* through size change of variables. We accurately track the size change of y in the first loop by transferring the potential 0.1 from $|[y,z]|$ to $|[0,y]|$. Furthermore, *t08a* shows again that we do not handle the constants 1 or 0 in any special way. In all examples we could replace 0 and 1 with other constants like in the second loop and still derive a tight bound. Example *t27* in Figure 2 shows how amortization can be used to handle *interacting nested loops*. In the outer loop we increment the variable n until $n = 0$. In each of the $|[n,0]|$ iterations, we increment the variable y by 1000. Then we non-deterministically (expressed by $*$) execute an inner loop that decrements y by 100 until $y < 100$. The analysis discovers that only the first execution of the inner loop depends on the initial value of y. We again derive tight constant factors.

Mutually Recursive Functions. As mentioned, the analysis also handles advanced control flow like `break` and `return` statements, and mutual recursion. Example *t39* in Figure 3 contains two mutually-recursive functions with their automatically derived tick bounds. The function `c_down` decrements its first argument x until it reaches the second argument y. It then recursively calls the function `c_up`, which is dual to `c_down`. Here, we count up y by 2 and call `c_down`. C^4B is the only available system that computes a tight bound.

Compositionality. With two concrete examples from open-source projects we demonstrate that the compositionality of our method is indeed crucial in practice.

Example *t61* in Figure 3 is typical for implementations of block-based cryptographic primitives: Data of arbitrary length is consumed in blocks and the leftover is stored in a buffer for future use when more data is available. It is present in all the block encryption routines of PGP and also used in performance critical code to unroll a loop. For example we found it in a bit manipulating function of the libtiff library and a CRC computation routine of MAD, an MPEG decoder. This looping pattern is handled particularly well by our method. If $N \geqslant 8$, C^4B infers the bound $\frac{N}{8}|[0,l]|$, but if $N < 8$, it infers $7\frac{8-N}{8} + \frac{N}{8}|[0,l]|$. The selection of the block size (8) and the cost in the second loop (`tick(1)`) are random choices and C^4B would also derive tight bound for other values.

To understand the resource bound for the case $N < 8$, first note that the cost of the second loop is $|[0, l]|$. After the first loop, we still have $\frac{N}{8}|[0, l]|$ potential available from the invariant. So we have to raise the potential of $|[0, l]|$ from $\frac{N}{8}$ to 1, that is, we must pay $\frac{8-N}{8}|[0, l]|$. But since we got out of the first loop, we know that $l < 8$, so it is sound to only pay $7\frac{8-N}{8}$ potential units instead. This level of precision and compositionality is only achieved by our novel analysis, no other available tool derives the aforementioned tight bounds.

Example *t62* (Figure 3) is the inner loop of a quick sort implementation in cBench. More precisely, it is the partitioning part of the algorithm. This partition loop has linear complexity, and feeding it to our analysis gives the worst-case bound $2 + 3|[l, h]|$. This bound is not optimal but it can be refined by rewriting the program. To understand the bound, we can reason as follows. If $h \geqslant l$ initially, the cost of the loop is 2. Otherwise, the cost of each round (at most 3) can be payed using the potential of $[l, h]$ by the first increment to l because we know that $l < h$. The two inner loops can also use $[l, h]$ to pay for their inner costs. KoAT fails to find a bound and LOOPUS derives the quadratic bound $(h - l - 1)^2$. Following the classical technique, these tools try to find one ranking function for each loop and combine them multiplicatively or additively.

In the extended version [14] is a list of more than 30 classes of challenging programs that we can automatically analyze. Section 8 contains a more detailed comparison with other tools.

4. Derivation System

In the following we describe the local and compositional derivation rules of the automatic amortized analysis.

Cost Aware Clight. We present the rules for a subset of Clight. Clight is the first intermediate language of the CompCert compiler [29]. It is a subset of C with a unified looping construct and side-effect free expressions. We reuse most of CompCert's syntax but instrument the semantics with a resource metric M that accounts for the cost (an arbitrary rational number) of each step in the operational semantics. For example, $M_e(\mathsf{exp})$ is the cost of evaluating the expression exp. The rationals M_f and M_r account respectively for the cost of a call to the function f and the cost of returning from it. More details are provided in Section 7.

In the rules, assignments are restricted to the form $x \leftarrow y$ or $x \leftarrow x \pm y$. In the implementation, a Clight program is converted into this form prior to analysis without changing the resource cost. This is achieved by using a series of *cost-free assignments* that do not result in additional cost in the semantics. Non-linear operations such as $x \leftarrow z * y$ or $x \leftarrow a[y]$ are handled by assigning 0 to coefficients like q_{xa} and q_{ax} that contain x after the assignment. This sound treatment ensures that no further loop bounds depend on the result of the non-linear operation.

Judgements. The derivation system for the automatic amortized analysis is defined in Figure 4. The derivation rules derive judgements of the form

$$(\Gamma_B; Q_B), (\Gamma_R; Q_R) \vdash \{\Gamma; Q\}\, S\, \{\Gamma'; Q'\}.$$

The part $\{\Gamma; Q\}\, S\, \{\Gamma'; Q'\}$ of the judgement can be seen as a quantitative Hoare triple. All assertions are split into two parts, the logical part and the quantitative part. The quantitative part Q represents a potential function as a collection of non-negative numbers q_i indexed by the index set I. The logical part Γ is left abstract but is enforced by our derivation system to respect classic Hoare logic constraints. The meaning of this basic judgment is as follows: If S is executed with starting state σ, the assertions in Γ hold, and at least $Q(\sigma)$ resources are available then the evaluation does not run out of resources and, if the execution terminates in state σ', there are at least $Q'(\sigma')$ resources left and Γ' holds for σ'.

The judgement is a bit more involved since we have to take into account the early exit statements break and return. This is similar to classical Hoare triples in the presence of non-linear control flow. In the judgement, $(\Gamma_B; Q_B)$ is the postcondition that holds when breaking out of a loop using break. Similarly, $(\Gamma_R; Q_R)$ is the postcondition that holds when returning from a function call.

As a convention, if Q and Q' are quantitative annotations we assume that $Q = (q_i)_{i \in I}$ and $Q' = (q'_i)_{i \in I}$. The notation $Q \pm n$ used in many rules defines a new context Q' such that $q'_0 = q_0 \pm n$ and $\forall i \neq 0.\, q'_i = q_i$. In all the rules, we have the implicit side condition that all rational coefficients are non-negative. Finally, if a rule mentions Q and Q' and leaves the latter undefined at some index i we assume that $q'_i = q_i$.

Function Specifications. During the analysis, function specifications are quadruples $(\Gamma_f; Q_f, \Gamma'_f; Q'_f)$ where $\Gamma_f; Q_f$ depend on $a\vec{r}gs$, and $\Gamma'_f; Q'_f$ depend on ret. These parameters are instantiated by appropriate variables on call sites. A distinctive feature of our analysis is that it respects the function abstraction: when deriving a function specification it generates a set of constraints and the above quadruple; once done, the constraint set can readily be reused for every call site and the function need not be analyzed multiple times. Therefore, the derivation rules are parametric in a function context Δ that we leave implicit in the rules presented here. More details can be found in the extended version.

Derivation Rules. The rules of our derivation system must serve two purposes. They must *attach* potential to certain program variable intervals and use this potential, when it is allowed, to *pay* for resource consuming operations. These two purposes are illustrated on the Q:SKIP rule. This rule reuses its precondition as postcondition, it is explained by two facts: First, no resource is consumed by the skip operation, thus no potential has to be used to pay for the evaluation. Second, the program state is not changed by the execution of a skip statement. Thus all potential available before the execution of the skip statement is still available after.

The rules Q:INCP, Q:DECP, and Q:INC describe how the potential is distributed after a size change of a variable. The rule Q:INCP is for increments $x \leftarrow x + y$ and Q:DECP is for decrements $x \leftarrow x - y$. They both apply only when we can deduce from the logical context Γ that $y \geqslant 0$. Of course, there are symmetrical rules Q:INCN and Q:DECN (not presented here) that can be applied if y is negative. The rules are all equivalent in the case where $y = 0$. The rule Q:INC can be applied if we cannot find the sign of y.

To explain how rules for increment and decrement work, it is sufficient to understand the rule Q:INCP. The others follow the same idea and are symmetrical. In Q:INCP, the program updates a variable x with $x + y$ where $y \geqslant 0$. Since x is changed, the quantitative annotation must be updated to reflect the change of the program state. We write x' for the value of x after the assignment. Since x is the only variable changed, only intervals of the form $[u, x]$ and $[x, u]$ will be resized. Note that for any u, $[x, u]$ will get smaller with the update, and if $x' \in [x, u]$ we have $|[x, u]| = |[x, x']| + |[x', u]|$. But $|[x, x']| = |[0, y]|$ which means that the potential q'_{0y} in the postcondition can be increased by q_{xu} under the guard that $x' \in [x, u]$. Dually, the interval $[v, x]$ can get bigger with the update. We know that $|[v, x']| \leqslant y + |[v, x]|$. So we decrease the potential of $[0, y]$ by q_{vx} to pay for this change. The rule ensures this only for $v \notin \mathcal{U}$ because $x \leqslant v$ otherwise, and thus $|[v, x]| = 0$.

The rule Q:LOOP is a cornerstone of our analysis. To apply it on a loop body, one needs to find an invariant potential Q that will pay for the iterations. At each iteration, M_l resources are spent to jump back. This explains the postcondition $Q + M_l$. Since the loop can only be exited with a break statement, the postcondition $\{\Gamma'; Q'\}$ for

$$\frac{}{B, R \vdash \{\Gamma; Q\}\ \mathsf{skip}\ \{\Gamma; Q\}}\ (\text{Q:SKIP}) \qquad \frac{}{B, R \vdash \{\Gamma; Q{+}M_a\}\ \mathsf{assert}\ e\ \{\Gamma \wedge e; Q\}}\ (\text{Q:ASSERT})$$

$$\frac{}{B, R \vdash \{\Gamma; Q{+}M_t(n)\}\ \mathsf{tick}(n)\ \{\Gamma; Q\}}\ (\text{Q:TICK}) \qquad \frac{}{(\Gamma; Q_B), R \vdash \{\Gamma; Q_B{+}M_b\}\ \mathsf{break}\ \{\Gamma'; Q'\}}\ (\text{Q:BREAK})$$

$$\frac{P = Q_R[ret/x] \qquad \Gamma = \Gamma_R[ret/x] \qquad \forall i \in \mathrm{dom}(P).\ p_i = q_i}{B, (\Gamma_R; Q_R) \vdash \{\Gamma; Q\}\ \mathsf{return}\ x\ \{\Gamma'; Q'\}}\ (\text{Q:RETURN}) \qquad \frac{(\Gamma'; Q'), R \vdash \{\Gamma; Q\}\ S\ \{\Gamma; Q{+}M_l\}}{B, R \vdash \{\Gamma; Q\}\ \mathsf{loop}\ S\ \{\Gamma'; Q'\}}\ (\text{Q:LOOP})$$

$$\frac{B, R \vdash \{\Gamma; Q\}\ S_1\ \{\Gamma'; Q'{+}M_s\} \qquad B, R \vdash \{\Gamma'; Q'\}\ S_2\ \{\Gamma''; Q''\}}{B, R \vdash \{\Gamma; Q\}\ S_1; S_2\ \{\Gamma''; Q''\}}\ (\text{Q:SEQ}) \qquad \frac{\begin{array}{c}B, R \vdash \{\Gamma \wedge e; Q{-}M_c^1)\}\ S_1\ \{\Gamma'; Q'\} \\ B, R \vdash \{\Gamma \wedge \neg e; Q{-}M_c^2\}\ S_2\ \{\Gamma'; Q'\}\end{array}}{B, R \vdash \{\Gamma; Q{+}M_e(e)\}\ \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2\ \{\Gamma'; Q'\}}\ (\text{Q:IF})$$

$$\frac{\begin{array}{c}\Gamma \models y \geqslant 0 \qquad \mathcal{U} = \{u \mid \Gamma \models x + y \in [x, u]\} \\ q'_{0y} = q_{0y} + \sum_{u \in \mathcal{U}} q_{xu} - \sum_{v \notin \mathcal{U}} q_{vx}\end{array}}{B, R \vdash \{\Gamma[x/x{+}y]; Q{+}M_u{+}M_e(x{+}y)\}\ x \leftarrow x + y\ \{\Gamma; Q'\}}\ (\text{Q:INCP}) \qquad \frac{\begin{array}{c}M = M_u + M_e(x \pm y) \\ q'_{0y} = q_{0y} - \sum_v q_{vx} \qquad q'_{y0} = q_{y0} - \sum_v q_{xv}\end{array}}{B, R \vdash \{\Gamma[x/x \pm y]; Q{+}M\}\ x \leftarrow x \pm y\ \{\Gamma; Q'\}}\ (\text{Q:INC})$$

$$\frac{\begin{array}{c}q'_{xy}, q'_{yx} \in \mathbb{Q}_0^+ \\ \forall u.(q_{yu} = q'_{xu} + q'_{yu} \wedge q_{uy} = q'_{ux} + q'_{uy})\end{array}}{B, R \vdash \{\Gamma[x/y]; Q{+}M_u{+}M_e(y)\}\ x \leftarrow y\ \{\Gamma; Q'\}}\ (\text{Q:SET}) \qquad \frac{\begin{array}{c}\Gamma \models y \geqslant 0 \qquad \mathcal{U} = \{u \mid \Gamma \models x - y \in [u, x]\} \\ q'_{y0} = q_{y0} + \sum_{u \in \mathcal{U}} q_{ux} - \sum_{v \notin \mathcal{U}} q_{xv}\end{array}}{B, R \vdash \{\Gamma[x/x{-}y]; Q{+}M_u{+}M_e(x{-}y)\}\ x \leftarrow x - y\ \{\Gamma; Q'\}}\ (\text{Q:DECP})$$

$$\frac{\begin{array}{c}(\Gamma_f; Q_f, \Gamma'_f; Q'_f) \in \Delta(f) \qquad \mathrm{Loc} = \mathrm{Locals}(Q) \qquad \forall i \neq j.\ x_i \neq x_j \qquad c \in \mathbb{Q}_0^+ \qquad Q = P + S \qquad Q' = P' + S \qquad U = Q_f[a\vec{r}gs/\vec{x}] \\ U' = Q'_f[ret/r] \qquad \forall i \in \mathrm{dom}(U).\ p_i = u_i \qquad \forall i \in \mathrm{dom}(U').\ p'_i = u'_i \qquad \forall i \notin \mathrm{dom}(U').\ p'_i = 0 \qquad \forall i \notin \mathrm{Loc}.\ s_i = 0\end{array}}{B, R \vdash \{\Gamma_f[a\vec{r}gs/\vec{x}] \wedge \Gamma_{\mathrm{Loc}}; Q{+}c{+}M_f\}\ r \leftarrow f(\vec{x})\ \{\Gamma'_f[ret/r] \wedge \Gamma_{\mathrm{Loc}}; Q'{+}c{-}M_r\}}\ (\text{Q:CALL})$$

$$\frac{\begin{array}{c}\Sigma f = (\vec{y}, S_f) \\ B, (\Gamma'_f; Q'_f) \vdash \{\Gamma_f[a\vec{r}gs/\vec{y}]; Q_f[a\vec{r}gs/\vec{y}]\}\ S_f\ \{\Gamma'; Q'\}\end{array}}{(\Gamma_f; Q_f, \Gamma'_f; Q'_f) \in \Delta(f)}\ (\text{Q:EXTEND}) \qquad \frac{\begin{array}{c}B, R \vdash \{\Gamma_2; Q_2\}\ S\ \{\Gamma'_2; Q'_2\} \qquad \Gamma_1 \models \Gamma_2 \\ Q_1 \geq_{\Gamma_1} Q_2 \qquad \Gamma'_2 \models \Gamma'_1 \qquad Q'_2 \geq_{\Gamma'_2} Q'_1\end{array}}{B, R \vdash \{\Gamma_1; Q_1\}\ S\ \{\Gamma'_1; Q'_1\}}\ (\text{Q:WEAK})$$

$$\frac{\begin{array}{c}\mathcal{L} = \{xy \mid \exists l_{xy} \in \mathbb{N}.\ \Gamma \models l_{xy} \leqslant |[x, y]|\} \qquad \mathcal{U} = \{xy \mid \exists u_{xy} \in \mathbb{N}.\ \Gamma \models |[x, y]| \leqslant u_{xy}\} \\ \forall i \in \mathcal{U}.\ q'_i \geqslant q_i - r_i \qquad \forall i \in \mathcal{L}.\ q'_i \geqslant q_i + p_i \qquad \forall i \notin \mathcal{U} \cup \mathcal{L} \cup \{0\}.\ q'_i \geqslant q_i \qquad q'_0 \geqslant q_0 + \sum_{i \in \mathcal{U}} u_i r_i - \sum_{i \in \mathcal{L}} l_i p_i\end{array}}{Q' \geq_\Gamma Q}\ (\text{RELAX})$$

Figure 4. Inference rules of the quantitative analysis.

the statement $\mathsf{loop}\ S$ is used as break postcondition in the derivation for S.

Another interesting rule is Q:CALL. It needs to account for the changes to the stack caused by the function call, the arguments/return value passing, and the preservation of local variables. We can sum up the main ideas of the rule as follows.

- The potential in the pre- and postcondition of the function specification is equalized to its matching potential in the callee's pre- and postcondition.
- The potential of intervals $|[x, y]|$ is preserved across a function call if x and y are local.
- The unknown potentials after the call (e.g. $|[x, g]|$, with x local and g global) are set to zero in the postcondition.

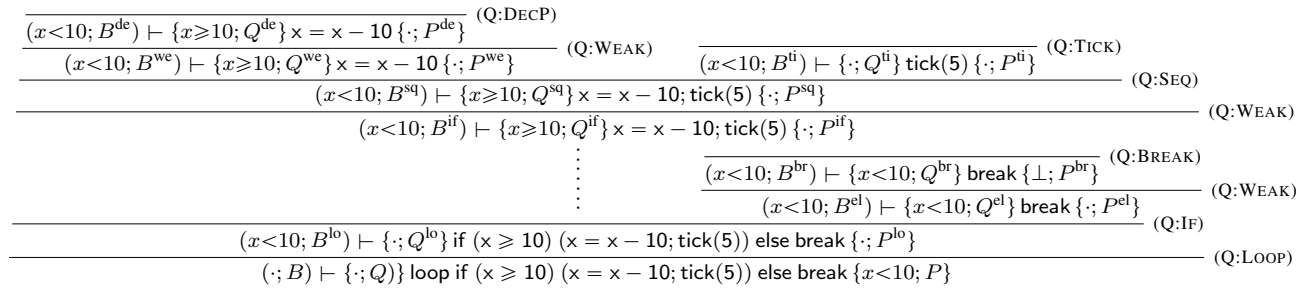
If x and y are local variables and $f(x, y)$ is called, Q:CALL splits the potential of $|[x, y]|$ in two parts. One part to perform the computation in the function f and one part to keep for later use after the function call. This splitting is realized by the equations $Q = P{+}S$ and $Q' = P'{+}S'$. Arguments in the function precondition $(\Gamma_f; Q_f)$ are named using a fixed vector $a\vec{r}gs$ of names different from all program variables. This prevents name conflicts and ensures that the substitution $[a\vec{r}gs/\vec{x}]$ is meaningful. Symmetrically, we use the unique name ret to represent the return value in the function's postcondition $(\Gamma'_f; Q'_f)$.

The rule Q:WEAK is the only rule that is not syntax directed. We could integrate weakenings into every syntax directed rule but, for the sake of efficiency, the implementation uses a simple heuristic instead. The high-level idea of Q:WEAK is the following: If we have a sound judgement, then it is sound to add more potential to the precondition and remove potential from the postcondition. The concept of *more potential* is formalized by the relation $Q' \geq_\Gamma Q$ that is defined in the rule RELAX. This rule also deals with the important task of transferring constant potential (represented by q_0) to interval sizes and vice versa. If we can deduce from the logical context that the interval size $|[x, y]| \geqslant \ell$ is larger than a constant ℓ then we can turn the potential $q_{xy} \cdot |[x, y]|$ form the interval into the constant potential $\ell \cdot q_{xy}$ and guarantee that we do not gain potential. Conversely, if $|[x, y]| \leqslant u$ for a constant u then we can transfer constant potential $u \cdot q_{xy}$ to the interval potential $q_{xy} \cdot |[x, y]|$ without gaining potential.

5. Automatic Inference via LP Solving

We separate the search of a derivation in two steps. As a first step we go through the functions of the program and apply inductively the derivation rules of the automatic amortized analysis. This is done in a bottom-up way for each strongly connected component (SCC) of the call graph. During this process our tool uses symbolic names for the rational coefficients q_i in the rules. Each time a linear constraint must be satisfied by these coefficients, it is recorded in a global list for the SCC using the symbolic names. We reuse the constraint list for every call from outside the SCC.

We then feed the collected constraints to an off-the-shelf LP solver (currently CLP [17]). If the solver successfully finds a solution, we know that a derivation exists and extract the values for the initial Q from the solver to get a resource bound for the program. To get a full derivation, we extract the complete solution from the



Constraints:

$P{=}B^{\text{lo}} \wedge Q{=}Q^{\text{lo}}{=}P^{\text{lo}}$ $\quad B^{\text{el}}{=}B^{\text{if}}{=}B^{\text{lo}} \wedge Q^{\text{el}}{=}Q^{\text{if}}{=}Q^{\text{lo}} \wedge P^{\text{el}}{=}P^{\text{if}}{=}P^{\text{lo}}$ $\quad B^{\text{el}}{=}B^{\text{br}} \wedge Q^{\text{el}} \geq_{(x<10)} Q^{\text{br}} \wedge P^{\text{br}} \geq_{(\cdot)} P^{\text{el}}$

$B^{\text{br}}{=}Q^{\text{br}}$ $\quad B^{\text{if}}{=}B^{\text{sq}} \wedge Q^{\text{if}} \geq_{(x<10)} Q^{\text{sq}} \wedge P^{\text{sq}} \geq_{(\cdot)} P^{\text{if}}$ $\quad B^{\text{sq}}{=}B^{\text{we}}{=}B^{\text{ti}} \wedge Q^{\text{sq}}{=}Q^{\text{we}} \wedge P^{\text{we}}{=}Q^{\text{ti}} \wedge P^{\text{ti}}{=}P^{\text{sq}}$

$Q^{\text{ti}}{=}P^{\text{ti}} + 5$ $\quad B^{\text{we}}{=}B^{\text{de}} \wedge Q^{\text{we}} \geq_{(x<10)} Q^{\text{de}} \wedge P^{\text{de}} \geq_{(\cdot)} P^{\text{we}}$ $\quad p^{\text{de}}_{0,10}{=}q^{\text{de}}_{0,10} + q^{\text{de}}_{0,x} \wedge p^{\text{de}}_{0}{=}q^{\text{de}}_{0} \wedge \forall(\alpha,\beta) \neq (0,10).\, p^{\text{de}}_{\alpha,\beta}{=}q^{\text{de}}_{\alpha,\beta}$

Linear Objective Function: $1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x}$ Constant Objective Function: $1{\cdot}q_0 + 11{\cdot}q_{0,10}$

Figure 5. An example derivation as produced C^4B. The constraints are resolved by an off-the-shelf LP solver.

solver and apply it to the symbolic names q_i of the coefficients in the derivation. If the LP solver fails to find a solution, an error is reported.

Figure 5 contains an example derivation as produced by C^4B. The upper case letters (with optional superscript) such as Q^{de} are families of variables that are later part of the constraint system that is passed to the LP solver. For example Q^{de} stands for the potential function $q^{\text{de}}_0 + q^{\text{de}}_{x,0}|[x,0]| + q^{\text{de}}_{0,x}|[0,x]| + q^{\text{de}}_{x,10}|[x,10]| + q^{\text{de}}_{10,x}|[10,x]| + q^{\text{de}}_{0,10}|[0,10]|$, where the variables such as $q^{\text{de}}_{x,10}$ are yet unknown and later instantiated by the LP solver.

In general, the weakening rule can be applied after every syntax directed rule. However, it can be left out in practice at some places to increase the efficiency of the tool. The weakening operation $\geq_\Gamma$ is defined by the rule RELAX. It is parameterized by a logical context that is used to gather information on interval sizes. For example,

$$\begin{aligned} P^{\text{de}} \geq_{(\cdot)} P^{\text{we}} \equiv\ & p^{\text{we}}_{0,10} \leqslant p^{\text{de}}_{0,10} + u_{0,10} - v_{0.10} \\ & \wedge\ p^{\text{we}}_{0} \leqslant p^{\text{de}}_{0} - 10{\cdot}u_{0,10} + 10{\cdot}v_{0.10} \\ & \wedge\ \forall(\alpha,\beta) \neq (0,10).\, p^{\text{we}}_{\alpha,\beta} \leqslant p^{\text{de}}_{\alpha,\beta}\ . \end{aligned}$$

The other rules are syntax directed and applied inductively. For example, the outermost expression is a loop, so we use the rule Q:Loop at the root of the derivation tree. At this point, we do not know yet whether a loop invariant exists. But we produce the constraints $Q^{\text{lo}} = P^{\text{lo}}$. These constraints express the fact that the potential functions before and after the loop body are equal and thus constitute an invariant.

After the constraint generation, the LP solver is provided with an objective function to be minimized. We wish to minimize the initial potential, which is a resource bound on the whole program. Here it is given by Q. Moreover, we would like to express that minimization of linear potential such as $q_{10,x}|[10,x]|$ takes priority over minimization of constant potential such as $q_{0,10}|[0,10]|$.

To get a tight bound, we use modern LP solvers that allow constraint solving and minimization at the same time: First we consider our initial constraint set as given in Figure 5 and ask the solver to find a solution that satisfies the constraints and minimizes the linear expression $1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x}$. The penalties given to certain factors are used to prioritize certain intervals. For example, a bound with $[10,x]$ will be preferred to another with $[0,x]$ because $|[10,x]| \leqslant |[0,x]|$. The LP solver now returns a solution of the constraint set and an objective value. The solver also memorizes the optimization path that led to the optimal solution. In this case, the objective value would be 5000 since the LP solver assigns $q_{0,x} = 0.5$ and $q_* = 0$ otherwise. We now add the constraint $1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x} \leqslant 5000$ to our constraint set and ask the solver to optimize the objective function $q_0 + 11{\cdot}q_{0,10}$. This happens in almost no time in practice. The final solution is $q_{0,x} = 0.5$ and $q_* = 0$ otherwise. Thus the derived bound is $0.5|[0,x]|$.

A notable advantage of the LP-based approach compared to SMT-solver–based techniques is that a satisfying assignment is a proof certificate instead of a counter example. To provide high-assurance bounds, this certificate can be checked in linear time by a simple validator.

6. Logical State and User Interaction

While complete automation is desirable, it is not always possible since the problem of bound derivation is undecidable. In this section we present a new technique to derive complex resource bounds semi-automatically by leveraging our automation. Our goal is to develop an interface between bound derivation and established qualitative verification techniques.

When the resource bound of a program depends on the contents of the heap, or is non-linear (e.g. logarithmic, exponential), we introduce a *logical state* using *auxiliary variables*. Auxiliary variables guide C^4B during bound derivation but they do not change the behavior of the program.

More precisely, the technique consists of the following steps. First, a program P that fails to be analyzed automatically is enriched by auxiliary variables $\vec{x}$ and assertions to form a program $P_l(\vec{x})$. Second, an initial value $\vec{X}(\sigma)$ for the logical variables is selected to satisfy the proposition:

$$\forall n\, \sigma\, \sigma'.\, (\sigma, P_l(\vec{X}(\sigma))) \Downarrow_n \sigma' \Longrightarrow \exists n' \leqslant n.\, (\sigma, P) \Downarrow_{n'} \sigma'. \quad (*)$$

Since the annotated program and the original one are usually syntactically close, the proof of this result goes by simple induction on the resource-aware evaluation judgement. Third, using existing automation tools, a bound $B(\vec{x})$ for $P_l(\vec{x})$ is derived. Finally this bound, instantiated with $\vec{X}$, gives the final resource bound for the program P.

This idea is illustrated by the program in Figure 6. The parts of the code in blue are annotations that were added to the original program text. The top-level loop increments a binary counter k times. A naive analysis of the algorithm yields the quadratic bound $k \cdot N$. However, the algorithm is in fact linear and its cost is bounded

```
1  logical state invariant {na = #1(a)}
2  while (k > 0) {
3    x=0;
4    while (x < N && a[x] == 1) {
5      assert(na > 0);
6      a[x]=0; na--;
7      tick(1); x++; }
8    if (x < N) { a[x]=1; na++; tick(1); }
9    k--;
10 }
```

Figure 6. Assisted bound derivation using logical state. We write $\#_1(a)$ for $\#\{i \mid 0 \leqslant i < N \wedge a[i]=1\}$ and use the tick metric. The derived bound is $2|[0,k]| + |[0,na]|$.

```
1  logical state invariant {lg > log2(h − l)}
2  bsearch(x,l,h,lg) {
3    if (h-l > 1) {
4      assert(lg > 0);
5      m = l + (h-l)/2;
6      lg--; if (a[m]>x) h=m; else l=m;
7      tick(M_bsearch);
8      l = bsearch(x,l,h,lg);
9      tick(−M_bsearch);
10   } else return l;
11 }
```

Figure 7. Assisted bound derivation using logical state. We write $\log_2(x)$ for the integer part of logarithm of x in base 2. The semi-automatically derived bound is $|[0, lg]|$.

by $2k + \#_1(a)$ where $\#_1(a)$ denotes the number of one entries in the array a. Since this number depends on the heap contents, no tool available for C is able to derive the linear bound. However, it can be inferred by our automated tool if a logical variable na is introduced. This logical variable is a reification of the number $\#_1(a)$ in the program. For example, on line 6 of the example we are setting `a[x]` to 0 and because of the condition we know that this array entry was 1. To reflect this change on $\#_1(a)$, the logical variable na is decremented. Similarly, on line 8, an array entry which was 0 becomes 1, so na is incremented. To complete the step 2 of the systematic procedure described above, we must show that the extra assertion `na > 0` on line 5 cannot fail. We do it by proving inductively that $na = \#_1(a)$ and remarking that since `a[x] == 1` is true, we must have $\#_1(a) > 0$, thus the assertion `na > 0` never fails.

Another simple example is given in Figure 7 where a logarithmic bound on the stack consumption of a binary search program is proved using logical variable annotations. Once again, annotations are in blue in the program text. In this example, to ease the proof of equivalence between the annotated program and the original one, we use the inequality $lg > \log_2(h - l)$ as invariant. This allows a simpler proof because, when working with integer arithmetic, it is not always the case that $\log_2(x - x/2) = \log_2(x) - 1$.

Generally, we observed that because the instrumented program is structurally same as the original one, it is enough to prove that the added assertions never fail in order to show the two programs satisfy the proposition (*). This can usually be piggybacked on standard static-analysis tools.

7. Soundness Proof

The soundness of the analysis builds on a new cost semantics for Clight and an extended quantitative logic. Using these two tools, the soundness of the automatic analysis described in Section 3 is proved by a translation morphism to the logic.

The main parts of the soundness proof are formalized with Coq and available for download. The full definitions of the cost semantics and the quantitative Hoare logic, and more details on the soundness proof can be found in the extended version of this article.

Cost Semantics for Clight. To base the soundness proof on a formal ground, we start by defining a new cost-aware operational semantics for Clight. Clight's operational semantics is based on small-step transitions and continuations. Expressions—which do not have side effects—are evaluated in a big-step fashion.

A program state $\sigma = (\theta, \gamma)$ is composed of two maps from variable names to integers. The first map, θ : Locals $\to \mathbb{Z}$, assigns integers to local variables of a function, and the second map, γ : Globals $\to \mathbb{Z}$, gives values to global variables of the program. In this article, we assume that all values are integers but in the implementation we support all data types of Clight. The evaluation function $[\![\cdot]\!]$ maps an expression $e \in E$ to a value $[\![e]\!]_\sigma \in \mathbb{Z}$ in the program state σ. We write $\sigma(x)$ to obtain the value of x in program state σ. Similarly, we write $\sigma[x \mapsto v]$ for the state based on σ where the value of x is updated to v.

The small-step semantics is standard, except that it tracks the resource consumption of a program. The semantics is parametric in the resource of interest for the user of our system. We achieve this independence by parameterizing evaluations with a resource *metric* M; a tuple of rational numbers and two maps. Each of these parameters indicates the amount of resource consumed by a corresponding step in the semantics. Resources can be released by using a negative cost. Two sample rules for update and tick follow.

$$\frac{\sigma' = \sigma[x \mapsto [\![e]\!]_\sigma]}{(\sigma, x \leftarrow e, K, c) \to (\sigma', \mathsf{skip}, K, c - M_u - M_e(e))}\ (\mathrm{U}) \qquad \frac{}{(\sigma, \mathsf{tick}(n), K, c) \to (\sigma, \mathsf{skip}, K, c - M_t(n))}\ (\mathrm{T})$$

The rules have as implicit side condition that c is non-negative. This makes it possible to detect a resource crash as a stuck configuration where $c < 0$.

Quantitative Hoare Logic. To prove the soundness of C^4B we found it useful to go through an intermediate step using a quantitative Hoare logic. This logic is at the same time a convenient semantic tool and a clean way to interface manual proofs with our automation. We base it on a logic for stack usage [13], add support for arbitrary resources, and simplify the handling of auxiliary state.

We define quantitative Hoare triples as $B; R \vdash_L \{Q\}\, S\, \{Q'\}$ where B, R, Q, and Q' are maps from program states to an element of $\mathbb{Q}_0^+ \cup \{\infty\}$ that represents an amount of resources available. The assertions B and R are postconditions for the case in which the block S exits by a `break` or `return` statement. Additionally, R depends on the return value of the current function. The meaning of the triple $\{Q\}\, S\, \{Q'\}$ is as follows: If S is executed with starting state σ, the empty continuation, and at least $Q(\sigma)$ resource units available then the evaluation does not run out of resources and there are at least $Q'(\sigma')$ resources left if the evaluation terminates in σ'. The logic rules are similar to the ones in previous work and generalized to account for the cost introduced by our cost-aware semantics.

Finally, we define a strong compositional continuation-based soundness for triples and prove the validity of all the rules in Coq. The full version of this paper [14], provides explanations for the rules and a thorough overview of our soundness proof.

The Soundness Theorem. We use the quantitative logic as the target of a translation function for the automatic derivation system. This reveals two orthogonal aspects of the proof: on one side, it relies on amortized reasoning (the quantitative logic rules), and on the other side, it uses combinatorial properties of our linear potential functions (the automatic analysis rules).

Technically, we define a translation function $\mathcal{T}$ such that if a judgement J in the automatic analysis is derivable, $\mathcal{T}(J)$ is deriv-

	t09	t19	t30	t15	t13

	`i=1; j=0;`
	`while (j<x) {`
	`  j++;`
	`  if (i>=4)`
	`  i=1, tick(40);`
	`  else i++;`
	`  tick(1); }`
		`while (i>100) {`
		`  i--; tick(1);`
		`} i += k+50;`
		`while (i>=0) {`
		`  i--; tick(1);`
		`}`
			`while (x>0) {`
			`  x--;`
			`  t=x, x=y, y=t;`
			`  tick(1);`
			`}`
				`assert(y>=0);`
				`while (x > y) {`
				`  x -= y+1;`
				`  for (z=y; z>0; z--)`
				`  tick(1);`
				`  tick(1);`
				`}`
					`while (x>0) {`
					`  x--;`
					`  if (*) y++;`
					`  else`
					`  while (y>0)`
					`  y--, tick(1);`
					`  tick(1); }`

C^4B	$11\|[0,x]\|$	$50+\|[-1,i]\|+\|[0,k]\|$	$\|[0,x]\|+\|[0,y]\|$	$\|[0,x]\|$	$2\|[0,x]\|+\|[0,y]\|$
Rank	$23{\cdot}x-14$	$54+k+i$	—	$2+2x-y$	$0.5{\cdot}y^2{+}yx\ldots$
LOOPUS	$41\max(x,0)$	$\max(i{-}100,0)$ $+\max(k{+}i{+}51,0)$	—	—	$2\max(x,0)$ $+\max(y,0)$

Figure 8. Comparison of resource bounds derived by different tools on several examples with linear bounds.

able in the quantitative logic. By using $\mathcal{T}$ to translate derivations of the automatic analysis to derivations in the quantitative logic we can directly obtain a certified resource bound for the analyzed program.

The translation of an assertion $(\Gamma; Q)$ in the automatic analysis is defined by

$$\mathcal{T}(\Gamma; Q) := \lambda\sigma.\, \Gamma(\sigma) + \Phi_Q(\sigma),$$

where we write Φ_Q for the unique linear potential function defined by the quantitative annotation Q. The logical context Γ is implicitly lifted to a quantitative assertion by mapping a state σ to 0 if $\Gamma(\sigma)$ holds and to ∞ otherwise. These definitions let us translate the judgement $J := B, R \vdash \{P\}\, S\, \{P'\}$ by

$$\mathcal{T}(J) := \mathcal{T}(B); \mathcal{T}(R) \vdash_L \{\mathcal{T}(P)\}\, S\, \{\mathcal{T}(P')\}.$$

The soundness of the automatic analysis can now be stated formally with the following theorem.

Theorem 1 (Soundness of the automatic analysis)**.** *If J is a judgement derived by the automatic analysis, then $\mathcal{T}(J)$ is a quantitative Hoare triple derivable in the quantitative logic.*

The proof of this theorem is constructive and maps each rule of the automatic analysis directly to its counterpart in the quantitative logic. The trickiest parts are the translations of the rules for increments and decrements and the rule Q:WEAK for weakening because they make essential use of the algebraic properties of the potential functions.

8. Experimental Evaluation

We have experimentally evaluated the practicality of our automatic amortized analysis with more than 30 challenging loop and recursion patterns from open-source code and the literature [18–20]. A full list of examples is given in the extended version [14].

Figure 8 shows five representative loop patterns from the evaluation. Example *t09* is a loop that performs an expensive operation every 4 steps. C^4B is the only tool able to amortize this cost over the input parameter x. Example *t19* demonstrates the compositionality of the analysis. The program consists of two loops that decrement a variable i. In the first loop, i is decremented down to 100 and in the second loop i is decremented further down to -1. However, between the loops we assign `i += k+50`. So in total the program performs $52 + |[-1, i]| + |[0, k]|$ ticks. Our analysis finds this tight bound because our amortized analysis naturally takes into account the relation between the two loops. Example *t30* decrements both input variables x and y down to zero in an unconventional way. In the loop body, first x is decremented by one, then the values of the variables x and y are switched using the local variable t as a buffer. Our analysis infers the tight bound $|[0, x]| + |[0, y]|$. Sometimes we need some assumptions on the inputs in order to derive a bound. Example *t15* is such a case. We assume here that the input variable y is non-negative and write `assert(y>=0)`. The assignment `x -= y+1` in the loop is split in `x--` and `x -= y`. If we enter the loop then we know that $x > 0$, so we can obtain constant potential from `x--`. Then we know that $x \geqslant y \geqslant 0$, as a consequence we can share the potential of $|[0, x]|$ between $|[0, x]|$ and $|[0, y]|$ after `x -= y`.

Example *t13* shows how amortization can be used to find linear bounds for nested loops. The outer loop is iterated $|[0, x]|$ times. In the conditional, we either (the branching condition is arbitrary) increment the variable y or we execute an inner loop in which y is counted back to 0. C^4B computes a tight bound. The extended version also contains a discussion of the automatic bound derivation for the Knuth-Morris-Pratt algorithm for string search. C^4B finds the tight linear bound $1 + 2|[0, n]|$.

Table 1. Comparison of C^4B with other automatic tools.

	KoAT	Rank	LOOPUS	SPEED	C^4B
#bounds	9	24	20	14	32
#lin. bounds	9	21	20	14	32
#best bounds	0	0	11	14	29
#tested	14	33	33	14	33

To compare our tool with existing work, we focused on loop bounds and use a simple metric that counts the number of back edges (i.e., number of loop iterations) that are followed in the execution of the program because most other tools only bound this specific cost. In Figure 8, we show the bounds we derived (C^4B) together with the bounds derived by LOOPUS [31] and Rank [3]. We also contacted the authors of SPEED but have not been able to obtain this tool. KoAT [11] and PUBS [1] currently cannot operate on C code and the examples would need to be manually translated into a term-rewriting system to be analyzed by these tools. For Rank it is not completely clear how the computed bound relates to the C program since the computed bound is for transitions in an automaton that is derived from the C code. For instance, the bound $2 + y - x$ that is derived for *t08* only applies to the first loop in the program.

Table 1 summarizes the results of our experiments presented in Appendix A. It shows for each tool the number of derived bounds (#bounds), the number of asymptotically tight bounds (#lin. bounds), the number of bounds with the best constant factors in comparison with the other tools (#best bounds), and the number of examples that we were able to test with the tool (#tested). Since we were not able to run the experiments for KoAT and SPEED, we simply used the bounds that have been reported by the authors of the respective tools. The results show that our automatic amortized analysis outperforms the existing tools on our example programs. However, this experimental evaluation has to be taken with a grain of salt. Existing tools complement C^4B since they can derive polynomial bounds and support more features of C. We were particularly impressed by LOOPUS which is very robust, works on large C files, and derives very precise bounds.

Table 2 contains a compilation of the results of our experiments with the cBench benchmark suite. It shows a representative list of automatically derived function bounds. In total we analyzed more

Table 2. Derived bounds for functions from cBench.

Function	LoC	Bound	Time (s)
adpcm_coder	145	$1 + \|[0, \mathsf{N}]\|$	0.6
adpcm_decod	130	$1 + \|[0, \mathsf{N}]\|$	0.2
BF_cfb64_enc	151	$1 + 2\|[-1, \mathsf{N}]\|$	0.7
BF_cbc_enc	180	$2 + 0.25\|[-8, \mathsf{N}]\|$	1.0
mad_bit_crc	145	$61.19{+}0.19\|[-1, \mathsf{N}]\|$	0.4
mad_bit_read	65	$1 + 0.12\|[0, \mathsf{N}]\|$	0.05
MD5Update	200	$133.95{+}1.05\|[0, \mathsf{N}]\|$	1.0
MD5Final	195	141	0.22
sha_update	98	$2 + 3.55\|[0, \mathsf{N}]\|$	1.2
PackBitsDecode	61	$1 + 65\|[-129, \mathsf{cc}]\|$	0.6
KMPSearch	20	$1 + 2\|[0, n]\|$	0.1
ycc_rgb_conv	66	$\mathsf{nr} \cdot \mathsf{nc}$	0.1
uv_decode	31	$\log_2(\mathsf{UV\_NVS}) + 1$	0.1

than 2900 lines of code. In the LoC column we not only count the lines of the analyzed function but also the ones of all the function it calls. We analyzed the functions using a metric that assigns a cost 1 to all the back-edges in the control flow (loops, and function calls). The bounds for the functions ycc_rgb_conv and uv_decode have been inferred with user interaction as described in Section 6. The most challenging functions for C^4B have unrolled loops where many variables are assigned. This stresses our analysis because the number of LP variables has a quadratic growth in program variables. Even on these stressful examples, the analysis could finish in less than 2 seconds. For example, the sha_update function is composed of one loop calling two helper functions that in turn have 6 and 1 inner loops. In the analysis of the SHA algorithm, the compositionality of our analysis is essential to get a tight bound since loops on the same index are sequenced 4 and 2 times without resetting it. All other tools derive much larger constant factors.

With our formal cost semantics, we can run our examples for different inputs and measure the cost to compare it to our derived bound. Figure 9 shows such a comparison for Example *t08*, a variant of *t08a* from Section 3. One can see that the derived constant factors are the best possible if the input variable x is non-negative.

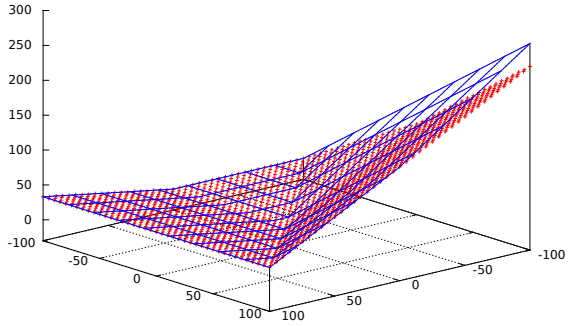


Figure 9. The automatically derived bound $1.33|[x, y]| + 0.33|[0, x]|$ (blue lines) and the measured runtime cost (red crosses) for Example *t08*. For $x \geqslant 0$ the bound is tight.

9. Limitations

Our implementation does not currently support all of Clight. Programs with function pointers, goto statements, continue statements, and pointers to stack-allocated variables cannot be analyzed automatically. While these limitations concern the current implementation, our technique is in principle capable to handle them.

For the sake of simplicity, the automated system described here is restricted to finding only linear bounds. However, the amortized analysis technique was shown to work with polynomial bounds [23]; we leave this extension of our system as future work.

Even certain linear programs cannot be analyzed automatically by C^4B, it is usually the case for programs that rely on heap invariants (like nul-terminated C strings), for programs in which resource usage depends on the result of non-linear operations (like $\%$ or $*$) in a non-trivial way, or for programs whose termination can only be proved by complex path-sensitive reasoning.

10. Related Work

Our work has been inspired by type-based amortized resource analysis for functional programs [21, 24, 25]. Here, we present the first automatic amortized resource analysis for C. None of the existing techniques can handle the example programs we describe in this work. The automatic analysis of realistic C programs is enabled by two major improvements over previous work. First, we extended the analysis system to associate potential with not just individual program variables but also multivariate intervals and, more generally, auxiliary variables. In this way, we solved the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on (possibly negative) integers without decreasing one individual number in each iteration. Second, for the first time, we have combined an automatic amortized analysis with a system for interactively deriving bounds. In particular, recent systems [22] that deal with integers and arrays cannot derive bounds that depend on values in mutable locations, possibly negative integers, or on differences between integers.

A recent project [13] has implemented and verified a quantitative logic to reason about stack-space usage, and modified the verified CompCert C compiler to translate C level bound to x86 stack bounds. This quantitative logic is also based on the potential method but has very rudimentary support for automation. It is not based on efficient LP solving and cannot automatically derive symbolic bounds. In contrast, our main contribution is an automatic amortized analysis for C that can derive parametric bounds for loops and recursive functions fully automatically. We use a more general quantitative Hoare logic that is parametric over the resource of interest.

There exist many tools that can automatically derive loop and recursion bounds for imperative programs such as SPEED [18, 20], KoAT [11], PUBS [1], Rank [3], ABC [8] and LOOPUS [31, 33]. These tools are based on abstract interpretation–based invariant generation and/or term rewriting techniques, and they derive impressive results on realistic software. The importance of amortization to derive tight bounds is well known in the resource analysis community [4, 27, 31]. Currently, the only other available tools that can be directly applied to C code are Rank and LOOPUS. As demonstrated, C^4B is more compositional than the aforementioned tools. Our technique, is the only one that can generate resource specifications for functions, deal with resources like memory that might become available, generate proof certificates for the bounds, and support user guidance that separates qualitative and quantitative reasoning.

There are techniques [10] that can compute the memory requirements of object oriented programs with region-based garbage collection. These systems can handle loops but not recursive or composed functions. We are only aware of two verified quantitative analysis systems. Albert et al. [2] rely on the KeY tool to automatically verify previously inferred loop invariants, size relations, and ranking functions for Java Card programs. However, they do not have a formal cost semantics and do not prove the bounds correct with respect to a cost model. Blazy et al. [9] have verified a loop bound analysis for CompCert's RTL intermediate language. However, this automatic bound analysis does not compute symbolic bounds.

11. Conclusion

We have developed a novel analysis framework for compositional and certified worst-case resource bound analysis for C programs. The framework combines ideas from existing abstract interpretation–

based techniques with the potential method of amortized analysis. It is implemented in the publicly available tool C^4B. To the best of our knowledge, C^4B is the first tool for C programs that automatically reduces the derivation of symbolic bounds to LP solving.

We have demonstrated that our approach improves the state-of-the-art in resource bound analysis for C programs in three ways. First, our technique is naturally compositional, tracks size changes of variables, and can abstractly specify the resource cost of functions (Section 3). Second, it is easily combinable with established qualitative verification to guide semi-automatic bound derivation (Section 6). Third, we have shown that the local inference rules of the derivation system automatically produce easily checkable certificates for the derived bounds (Section 7). Our system is the first amortized resource analysis for C programs. It addresses the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on signed integers and to deal with non-linear control flow.

This work is the starting point for several projects that we plan to investigate in the future, such as the extension to concurrency, better integration of low-level features like memory caches, and the extension of the automatic analysis to multivariate resource polynomials [23].

Acknowledgments

We thank members of the FLINT team at Yale and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by NSF grants 1319671 and 1065451, DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, and ONR Grant N00014-12-1-0478. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[2] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Fundamental Approaches to Software Engineering - 15th Int. Conf. (FASE'12)*, pages 130–145, 2012.

[3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.

[4] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th Int. Static Analysis Symposium (SAS'12)*, pages 405–421, 2012.

[5] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[6] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 90–101, 2009.

[7] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie. System-Level Non-Interference for Constant-Time Cryptography. *IACR Cryptology ePrint Archive*, 2014:422, 2014.

[8] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.

[9] S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th Int. Conf. (VSTTE'13)*, 2013. To appear.

[10] V. A. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *7th Int. Symp. on Memory Management (ISMM'08)*, pages 141–150, 2008.

[11] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

[12] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *28th Conf. on Object-Oriented Prog., Sys., Langs., and Appl., OOPSLA'13*, pages 33–52, 2013.

[13] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. In *Conf. on Prog. Lang. Design and Impl. (PLDI'14)*, page 30, 2014.

[14] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds (Extended Version). Technical Report YALEU/DCS/TR-1505, Dept. of Computer Science, Yale University, New Haven, CT, April 2015.

[15] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX Annual Technical Conference (USENIX'10)*, 2010.

[16] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy Types. In *27th Conf. on Object-Oriented Prog., Sys., Langs., and Appl., OOPSLA'12*, pages 831–850, 2012.

[17] COIN-OR Project. CLP (Coin-or Linear Programming). `https://projects.coin-or.org/Clp`, 2014. Accessed: 2014-11-12.

[18] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.

[19] S. Gulwani, S. Jain, and E. Koskinen. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*, pages 375–385, 2009.

[20] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[21] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[22] J. Hoffmann and Z. Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

[23] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symp. on Principles of Prog. Langs. (POPL'11)*, 2011.

[24] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[25] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[26] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006.

[27] M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Joint 25th RTA and 12th TLCA*, 2014.

[28] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Emb. Sys., 11th Int. Workshop (CHES'09)*, pages 1–17, 2009.

[29] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[30] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, 2005.

[31] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, page 743–759, 2014.

[32] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[33] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symposium (SAS'11)*, 2011.

A. Complete Experimental Results for the Tool Comparison

Table 3. Comparison of the bounds generated by KoAT, Rank, LOOPUS, SPEED, and our tool C^4B on several challenging linear examples. Results for KoAT and SPEED were extracted from previous publications [18–20, 31] because KoAT cannot take C programs as input in its current version and SPEED is not available. Entries marked with ? indicate that we cannot test the respective example with the tool. Entries marked with — indicate that the tool failed to produce a result. We write $\mathrm{mx}(a, b)$ for the maximum of a and b. Functions with names of the form tXX are challenging tests that we designed during the development of C^4B. The source code for all functions is available in the extended version [14].

Function	KoAT		Rank		LOOPUS		SPEED	C^4B
gcd	?		$(((2{+}1)\dots$	$O(n)$	—		?	$\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
kmp	?		$(((2{+}(n{+}\dots$	$O(n^2)$	$\mathrm{mx}(n,0)\dots$	$O(n)$	?	$1{+}2\lvert[0,n]\rvert$
qsort	?		—		—		?	$1{+}2\lvert[0,\mathsf{len}]\rvert$
speed pldi09 fig4 2	—		$(((2{+}n)\dots$	$O(n)$	—		$\frac{n}{m}+n$	$1{+}2\lvert[0,n]\rvert$
speed pldi09 fig4 4	—		$(((2{+}(-1\dots$	$O(n)$	—		$\frac{n}{m}+m$	$\lvert[0,n]\rvert$
speed pldi09 fig4 5	$28d+7g+27$	$O(n)$	$(((2{+}(-1\dots$	$O(n)$	—		$\mathrm{mx}(n,n-m)$	—
speed pldi10 ex1	—		—		—		n	$\lvert[0,n]\rvert$
speed pldi10 ex3	—		$(((2{+}(-1\dots$	$O(n)$	$2{\cdot}\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
speed pldi10 ex4	$110a+33$	$O(n)$	—		—		$n+1$	$1{+}2\lvert[0,n]\rvert$
speed popl10 fig2 1	$9a+9b+\dots$	$O(n)$	$((2{+}((-y\dots$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$\lvert[x,n]\rvert{+}\lvert[y,m]\rvert$
speed popl10 fig2 2	$6a{+}9b{+}3c+5$	$O(n)$	$((2{-}x\dots$	$O(n)$	$\mathrm{mx}(0,(x+1{-}z)\dots$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,n{-}z)$	$\lvert[x,n]\rvert{+}\lvert[z,n]\rvert$
speed popl10 nested multiple	—		$((2{-}x{+}n\dots$	$O(n^2)$	$\mathrm{mx}(0,m{-}y)+\mathrm{mx}(0,n{-}x)$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$\lvert[x,n]\rvert{+}\lvert[y,m]\rvert$
speed popl10 nested single	$48b+16$	$O(n)$	$(((1{-}x{+}n\dots$	$O(n)$	$\mathrm{mx}(0,n{-}1)\dots$	$O(n)$	n	$\lvert[0,n]\rvert$
speed popl10 sequential single	$21b+6$	$O(n)$	$((2-x{+}n\dots$	$O(n)$	$2{\cdot}\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
speed popl10 simple multiple	$9c+10d+7$	$O(n)$	$((2{-}y{+}m\dots$	$O(n)$	$\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$	$O(n)$	$n+m$	$\lvert[0,m]\rvert{+}\lvert[0,n]\rvert$
speed popl10 simple single2	$20d+12c+17$	$O(n)$	—		$\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$	$O(n)$	$n+m$	$\lvert[0,n]\rvert{+}\lvert[0,m]\rvert$
speed popl10 simple single	$4b+6$	$O(n)$	$((2{-}x{+}n\dots$	$O(n)$	$\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
t07	?		$2+x$	$O(n)$	$\mathrm{mx}(x,0)\dots$	$O(n)$	?	$1{+}3\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t08	?		$((2{+}z{-}y\dots$	$O(n)$	$\mathrm{mx}(0,y{-}2)\dots$	$O(n)$	?	$1.33\lvert[y,z]\rvert{+}0.33\lvert[0,y]\rvert$
t10	?		$((2{-}y{+}x\dots$	$O(n)$	$\mathrm{mx}(0,x{-}y)$	$O(n)$	?	$\lvert[y,x]\rvert$
t11	?		$((2{-}y{+}m\dots$	$O(n)$	$\mathrm{mx}(0,n{-}x))+\mathrm{mx}(0,m{-}y)$	$O(n)$	?	$\lvert[x,n]\rvert{+}\lvert[y,m]\rvert$
t13	?		$(((1{+}y^2/2\dots$	$O(n^2)$	$2{\cdot}\mathrm{mx}(x,0)+\mathrm{mx}(y,0)$	$O(n)$	?	$2\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t15	?		$((1{+}x\dots$	$O(n)$	—		?	$\lvert[0,x]\rvert$
t16	?		$((-99{\cdot}y\dots$	$O(n)$	—		?	$101\lvert[0,x]\rvert$
t19	?		$((153{+}k\dots$	$O(n)$	$\mathrm{mx}(0,i{-}10^2){+}\mathrm{mx}(0,k{+}i{+}51)$	$O(n)$	?	$50{+}\lvert[-1,i]\rvert{+}\lvert[0,k]\rvert$
t20	?		$(2{-}y{+}x\dots$	$O(n)$	$2{\cdot}\mathrm{mx}(0,y{-}x){+}\mathrm{mx}(0,x{-}y)$	$O(n)$	?	$\lvert[x,y]\rvert{+}\lvert[y,x]\rvert$
t27	?		—		$10^3\mathrm{mx}(0,-n)\dots$	$O(n)$	?	$0.01\lvert[n,y]\rvert{+}11\lvert[n,0]\rvert$
t28	?		$((1{-}y{+}x\dots$	$O(n)$	$10^3\,\mathrm{mx}(0,x-y)\dots$	$O(n)$	?	$\lvert[x,0]\rvert{+}\lvert[0,y]\rvert{+}1002\lvert[y,x]\rvert$
t30	?		—		—		?	$\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t37	?		—		—		?	$3{+}2\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t39	?		—		—		?	$1.33{+}0.67\lvert[z,y]\rvert$
t46	?		—		—		?	$\lvert[0,y]\rvert$
t47	?		$4+n$	$O(n)$	$1+\mathrm{mx}(n,0)$	$O(n)$	?	$1{+}\lvert[0,n]\rvert$