Selectively-Amortized Resource Bounding

Tianhan Lu, Bor-Yuh Evan Chang, Ashutosh Trivedi

University of Colorado Boulder
{tianhan.lu,bec,ashutosh.trivedi}@colorado.edu

Abstract. We consider the problem of automatically proving resource bounds. That is, we study how to prove that an integer-valued resource variable is bounded by a given program expression. Automatic resource-bound analysis has recently received significant attention because of a number of important applications (e.g., detecting performance bugs, preventing algorithmic-complexity attacks, identifying side-channel vulnerabilities), where the focus has often been on developing precise amortized reasoning techniques to infer the most exact resource usage. While such innovations remain critical, we observe that fully precise amortization is not always necessary to prove a bound of interest. And in fact, by amortizing *selectively*, the needed supporting invariants can be simpler, making the invariant inference task more feasible and predictable. We present a framework for selectively-amortized analysis that mixes worst-case and amortized reasoning via a property decomposition and a program transformation. We show that proving bounds in any such decomposition yields a sound resource bound in the original program, and we give an algorithm for selecting a reasonable decomposition.

1 Introduction

In recent years, automatic resource-bound analysis has become an increasingly specialized area of automated reasoning because of a number of important and challenging applications, including statically detecting performance bugs, preventing algorithmic-complexity attacks, and identifying side-channel vulnerabilities. In this paper, we consider the specific problem of proving bounds on resource usage as follows: given an integer-valued resource variable r that models resource allocation and deallocation, prove that it is bounded by an expression e_{ub} at any program location—that is, prove `assert` $r \leq e_{\text{ub}}$ anywhere in the program. Resource allocations and deallocations can be modeled by (ghost) updates `use` $r\ e_{\text{op}}$ to the resource variable r (expressing that resource usage captured by

r increments by e units), and we generically permit updates to be any expression e_{op}. For example, resource variables can model lengths of dynamically-sized collections like lists and strings (e.g., `List.size()` or `StringBuilder.length()` in Java), and resource updates capture growing or shrinking such collections (e.g., `List.add(Object)`, `List.remove(Object)`, or `StringBuilder.append(String)`).

There are two natural ways to address this problem, by analogy to amortized computational complexity [36], for which we give intuition here. The first approach views the problem as an extension of the loop bounding problem, that is, inferring an upper bound on the number of times a loop executes [8, 19, 20, 33–35, 39]. Then to derive upper bounds on resource variables r, multiply the worst-case, or upper bound, of an update expression e_{op} by an upper bound on the number of times that update is executed, summed over each resource-use command `use` r e_{op}, thereby leveraging the existing machinery of loop bound analysis [8, 9, 35]. We call this approach *worst-case reasoning*, as it considers the worst-case cost of a given resource-use command for each loop iteration. This worst-case reasoning approach has two potential drawbacks. First, it presupposes the existence of loop bounds (i.e., assumes terminating programs), whereas we may wish to prove resource usage remains bounded in non-terminating, reactive programs (e.g., Lu et al. [28]) or simply where loop bounds are particularly challenging to derive. Second, as the terminology implies, it can be overly pessimistic because the value of the resource-use expression e_{op} may vary across loop iterations.

The second approach to resource bound verification is to directly adopt the well-established method of finding inductive invariants strong enough to prove assertions [31]. However, directly applying inductive invariant inference techniques (e.g., Chatterjee et al. [12], Colón et al. [13], Dillig et al. [16], Hrushovski et al. [24], Kincaid et al. [25, 26, 27], Sharma et al. [32]) to the resource bounding can be challenging, because the required inductive invariants are often particularly complex (e.g., polynomial) and are thus not always feasible or predictable to infer automatically [9, 21]. We call this approach *fully-amortized reasoning*, as the strongest inductive invariant bounding the resource variable r may consider arbitrary relations to reason about how the resource-use expression e_{op} may vary across loop iterations, thereby reasoning about amortized costs across loop iterations.

The key insight of this paper is that the choice is not binary but rather the above two approaches are extremal instances on a spectrum of *selective amortization*. We can apply amortized reasoning within any sequence of resource updates and then reason about each sequence's contribution to the overall resource usage with worst-case reasoning. We show that the *decomposition* of the overall resource usage into *amortized segments* can be arbitrary, so it can be flexibly chosen to simplify inductive invariant inference for amortized reasoning of resources or to leverage loop bound inference where it is possible, easy, and precise. We then realize this insight through a program transformation that expresses a particular class of decompositions and enables using off-the-shelf amortized reasoning engines. In particular, we make the following contributions:

1. We define a space of amortized reasoning based on decomposing resource updates in different ways and then amortizing resource usage within the resulting segments (Section 3). Different decompositions *select* different amortizations, and we prove that any decomposition yields a sound upper bound.
2. We instantiate selective amortization through a program transformation for a particular class of decompositions and define a notion of *non-interfering amortization segments* to suggest a segmentation strategy (Section 4).
3. We implemented a proof-of-concept of selective amortization in a tool Brbo (for *br*eak-and-*bo*und) that selects a decomposition and then delegates to an off-the-shelf invariant generator for amortized reasoning (Section 5). Our empirical evaluation provides evidence that selective amortization effectively leverages both worst-case and amortized reasoning.

Our approach is agnostic to the underlying amortized reasoning engine. Directly applying a relational inductive invariant generator on resource variables, as we do in our proof-of-concept (Section 5), corresponds to an aggregate amortized analysis, however this work opens opportunities to consider other engines based on alternative amortized reasoning (e.g., the potential method [22, 23]).

2 Overview

Fig. 1 shows the core of Java template engine class from the DARPA STAC [15] benchmarks. The `replaceTags` method applies a list of templates `ts` to the input `text` using an intermediate `StringBuilder` resource `sb` that we wish to bound globally. In this section, we aim to show that proving such a bound on `sb` motivates selective amortized reasoning.

At a high-level, the `replaceTags` method allocates a fresh `StringBuilder sb` to copy non-tag text or to replace tags using the input templates `ts` from the input `text`. The inner loop at program point 4 does this copy or tag replacement by walking through the ordered list of tag locations `tags` to copy the successive "chunks" of non-tag text `text.substring(p, l)` and a tag replacement `rep` at program points 6 and 8, respectively (the `assume` statement at program point 5 captures the ordered list of locations property). Then, the leftover text `text.substring(p, text.length())` after the last tag is copied at program point 11. The outer loop at program point 2 simply does this template-based tag replacement, and inserts a separator `sep` (at program point 12), for each template `t`. There are four program points where resources of interest are *used* (i.e., `sb` grows in length)—the `sb.append(...)` call sites mentioned here.

The `@Bound` assertion shown on line 1

$$\texttt{\#sb} \leq \texttt{\#ts} \cdot (\texttt{\#text} + \texttt{\#tags} \cdot \texttt{ts\#rep} + \texttt{\#sep})$$

follows the structure of the code sketched above. The template-based tag replacement is done `#ts` number of times where `#ts` models the size of the template list `ts`. Then, the length of the tag-replaced text is bounded by the length of `text` (i.e., `#text`) plus a bound on the length of all tag-replaced text `#tags` $\cdot$ `ts#rep` plus

```
  private String text;
  private List<Pair<Integer,Integer>> tags = ...text...;
  public String replaceTags(List<Templated> ts, String sep) {
1   @Bound(#sb ≤ #ts·(#text + #tags·ts#rep + #sep)) StringBuilder
    sb = new StringBuilder();
2   for (Templated t : ts) {
3     int p = 0;
4     for (Pair<Integer,Integer> lr : tags) {
5       int l = lr.getLeft(); int r = lr.getRight();
          assume(p ≤ l ≤ r ≤ #text);
6       sb.append(text.substring(p, l));
7       String rep = ...t...lr...; assume(#rep ≤ ts#rep);
8       sb.append(rep);
9       p = r;
10    }
11    sb.append(text.substring(p, text.length()));
12    sb.append(sep);
13  }
    return sb.toString();
  }
```

Fig. 1: Motivating selective worst-case and amortized reasoning to analyze a Java template engine class (`com.cyberpointllc.stac.template.TemplateEngine`). An instance of this class stores some `text` that may have tags in it to replace with this engine. The tag locations are stored as an ordered list of pairs of start-end indexes in the `tags` field, which is computed from `text`. Suppose we want to globally bound the size of the `StringBulder` `sb` used by the `replaceTags` method to apply a list of templates `ts`. Let `#sb` be a resource variable modeling the length of `sb` (i.e., ghost state that should be equal to the run-time value of `sb.length()`). We express a global bound on `#sb` to prove with the `@Bound` annotation—here in terms of resource variables on the inputs `ts`, `text`, `tags`, and `sep`.

the length of the separator `sep` (i.e., `#sep`). A bound on each tag replacement `rep` is modeled with a variable `ts#rep` (which we name with `ts` to indicate its correspondence to a bound on all tag replacements described by input `ts`) and the `assume(#rep ≤ ts#rep)` statement at program point 7. Thus, a bound on the length of all tag-replaced text is `#tags · ts#rep`. Note that the coloring here is intended to ease tracking key variables but having color is not strictly necessary for following the discussion.

For explanatory purposes, the particular structure of this bound assertion also suggests a mix of worst-case and amortized reasoning that ultimately leads to our selectively-amortized reasoning approach that we describe further below. Starting from reasoning about the inner loop, to prove that the copying of successive "chunks" of `text` is bounded by `#text` requires amortized reasoning because the length of `text.substring(p, l)` at program point 6 varies on each loop iteration. In contrast, we bound the length of all tag-replaced text with

#tags · ts#rep using worst-case reasoning: we assume a worst-case bound on the length of replacement text rep is ts#rep, so a worst-case bound with #tags number of tag replacements is #tags · ts#rep. Now thinking about the rest of the body of the outer loop at program point 11, the leftover text copy is amortized with the inner loop's copying of successive "chunks," so we just add the length of the separator #sep. Finally, considering the outer loop, we simply consider this resource usage bound for each loop iteration to bound #sb with #ts·(...).

The key observation here is that to prove this overall bound on #sb, even though we need to amortize the calls to sb.append(text.substring(p, l)) at program point 6 over the iterations of the inner loop, we do not need to amortize the calls at this same site across iterations of the outer loop. Next, we translate this intuition into an approach for *selectively-amortized resource bounding*.

2.1 Decomposing Resource Updates to Selectively Amortize

The resource-bound reasoning from Fig. 1 may be similarly expressed in a numerical abstraction where all variables are of integer type as shown in Fig. 2a. There, we write use r e_{op} for tracking e_{op} units of resource use in r and x := $*$ for a havoc (i.e., a non-deterministic assignment). Note that text.substring(p, l) translates to (l - p). To express checking the global bound, we write assert(#sb $\leq e_{\text{ub}}$) after each use update. We also note a pre-condition that simply says that all of the inputs sizes are non-negative. Crucially, observe to precisely reason about the resource usage #sb across all of these updates to #sb requires a polynomial loop invariant, as shown at program point 5 in braces {···}.

Yet, our informal reasoning above did not require this level of complexity. The key idea is that we can conceptually decompose the intermingled resource updates to #sb in any number of ways—and different decompositions select different amortizations. In Fig. 2b, we illustrate a particular decomposition of updates to #sb. We introduce three resource variables #sb1, #sb2, #sb3 that correspond to the three parts of the informal argument above (i.e., resource use for the non-tag text at program points 6 and 11, the tag-replaced text at program point 8, and the separator at program point 12, respectively). Let us first ignore the reset and ub commands (described further below), then we see that we are simply accumulating resource updates to #sb into separate variables or *amortization groups* such that #sb = #sb1 + #sb2 + #sb3. But we can now bound #sb1, #sb2, and #sb3 independently and have the sum of the bounds of these variables be a bound for the original resource variable #sb.

However, precisely reasoning about the resource usage in #sb1 still requires a polynomial loop invariant with the loop counters i, input #text, and internal variable p. Following the observation from above, we want to amortize updates to #sb1 across iterations of the inner loop but not between iterations of the outer loop. That is, we want to amortize updates to #sb1 in the sequence of resource uses within a single iteration of the outer loop and then apply worst-case reasoning to the resource bound amortized within this sequence. The *amortization reset* reset #sb1 after the initializer of the loop at program point 4 accomplishes this desired decoupling of the updates to #sb1 between outer-loop iterations by

global bound e_{ub}: `#ts·(#text + #tags·ts#rep + #sep)`
pre-condition: $\{0 \le \texttt{\#text} \land 0 \le \texttt{\#tags} \land 0 \le \texttt{\#ts} \land 0 \le \texttt{ts\#rep} \land 0 \le \texttt{\#sep}\}$

```
1#sb := 0;
2for (i := 0; i < #ts; i++) {
3  p := 0;
4  for (j := 0
        ; j < #tags; j++) {
5   {#sb ≤ (i·#text+p)
         + ((i·#tags+j)·ts#rep)
         + (i·#sep)}
     l := *; r := *;
      assume(p ≤ l ≤ r ≤ #text);
6    use #sb (l - p);
      assert(#sb ≤ e_ub);
7    #rep := *;
      assume(0 ≤ #rep ≤ ts#rep);
8
     use #sb #rep;
      assert(#sb ≤ e_ub);
9    p := r;
10 }
11 use #sb (#text - p);
    assert(#sb ≤ e_ub);
12
   use #sb #sep;
    assert(#sb ≤ e_ub);
13}
```

(a) A numerical abstraction of the `replaceTags` method from Fig. 1.

```
1
2for (i := 0; i < #ts; i++) {
3  p := 0;
4  for (j := 0,
        reset #sb1; j < #tags; j++) {
5  {#sb1♯ = i ∧ #sb1* ≤ #text ∧ #sb1 ≤ p ∧
    #sb2♯ = i·#tags+j-1 ∧
    #sb2* ≤ ts#rep ∧ #sb2 ≤ ts#rep ∧
    #sb3♯ = i-1 ∧
    #sb3* ≤ #sep ∧ #sb3 ≤ #sep}
     l := *; r := *;
      assume(p ≤ l ≤ r ≤ #text);
6    use #sb1 (l - p);
      ub #sb1, #sb2, #sb3 e_ub
7    #rep := *;
      assume(0 ≤ #rep ≤ ts#rep);
8    reset #sb2;
     use #sb2 #rep;
      ub #sb1, #sb2, #sb3 e_ub
9    p := r;
10 }
11 use #sb1 (#text - p);
    ub #sb1, #sb2, #sb3 e_ub
12 reset #sb3;
   use #sb3 #sep;
    ub #sb1, #sb2, #sb3 e_ub
13}
```

(b) A resource usage decomposition and amortized segmentation of (a).

Fig. 2: Decomposing resource usage into amortized segments transforms the required supporting loop invariant at program point 5 needed to prove the global bound e_{ub} *from polynomial to linear.*

"resetting the amortization" at each outer-loop iteration. Conceptually, executions of the `reset` r mark the boundaries of the *amortization segments* of `use`s of resource r.

The result of this decomposition is the simpler invariant at program point 5 in the transformed program of Fig. 2b, which use some auxiliary summary variables like `#sb1`* and `#sb1`$^\sharp$. For every resource variable r, we consider two summary variables r^* and $r^\sharp$, corresponding, respectively, to the maximum of r in any segment and the number of "`reset`ted" r segments so far. Concretely, the semantics of `reset #sb1` is as follows: (1) increment the segment counter variable `#sb1`$^\sharp$ by

1, thus tracking the number of amortization segments of `#sb1 use`s; (2) bump up `#sb1`* if necessary (i.e., set `#sb1`* to `max(#sb1`*`, #sb1)`), tracking the maximum `#sb1` in any segment so far; and finally, (3) resets `#sb1` to 0 to start a new segment. As we see at program point 5 in the original and transformed programs of Fig. 2, we have decomposed the total non-tag text piece `(i·#text+p)` into `#sb1`$^\sharp \cdot$ `#sb1`$^*+$ `#sb1` where `#sb1`$^\sharp =$ `i`, `#sb1`$^* \leq$ `#text`, and `#sb1` $\leq$ `p`. Intuitively, `#sb1`$^\sharp \cdot$`#sb1`* upper-bounds the cost of all *past* iterations of the outer loop, and the cost of the *current* iteration is precisely `#sb1`. Thus, `#sb1`$^\sharp \cdot$ `#sb1`$^*+$ `#sb1` is globally and inductively an upper bound for the total non-tag text piece of `#sb`. The same decomposition applies to `#sb2` and `#sb3` where note that `#sb3`$^\sharp =$ `i-1`, counts past segments separated from the current segment so that `#sb3`$^\sharp \cdot$ `#sb3`$^*+$ `#sb3` corresponds to `(i·#sep)` where both the past and current are summarized together. Overall, combining the amortization groups and segments, we have the following global invariant between the original program and the transformed one:

$$\texttt{\#sb} \leq\ (\texttt{\#sb1}^\sharp \cdot \texttt{\#sb1}^* + \texttt{\#sb1}) + (\texttt{\#sb2}^\sharp \cdot \texttt{\#sb2}^* + \texttt{\#sb2}) + (\texttt{\#sb3}^\sharp \cdot \texttt{\#sb3}^* + \texttt{\#sb3})$$

To verify a given bound in the transformed program, we simply check that this expression on the right in the above is bounded by the desired bound expression using any inferred invariants on `#sb1`$^\sharp$, `#sb1`*, `#sb1`, etc. This is realized by the *upper-bound check* command at, for instance, program point 6 in Fig. 2b:

$$\texttt{ub\ \#sb1, \#sb2, \#sb3\ (\#ts·(\#text} + \texttt{\#tags·ts\#rep} + \texttt{\#sep))}\ .$$

Here, `ub` $\overline{r}$ e asserts that the sum of amortization groups (internally decomposed into amortization segments) in the set $\overline{r}$ is bounded from above by e.

2.2 Finding a Selective-Amortization Decomposition

Fig. 2b shows a decomposition of updates to `#sb` into groups (i.e., `#sb1`, `#sb2`, and `#sb3`) and segments (i.e., with `reset`s) that realize a particular selective amortization. We show that any decomposition into groups and segments is sound in Section 3, but here, we discuss how we find such a decomposition.

Intuitively, we want to use worst-case reasoning whenever possible, maximizing decoupling of updates and simplifying invariant inference. But some updates should be considered together for amortization. Thus, any algorithm to select a decomposition must attempt to resolve the tension between two conflicting goals: partitioning `use` updates in the program into more groups and smaller segments, but also allowing amortizing costs inside larger segments to avoid precision loss. For example, it is important to use the same accumulation variable `#sb1` for the two locations that contribute to the non-tag text (program points 6 and 11) to amortize over both `use` sites. In Section 4, we characterize the potential imprecision caused by worst-case reasoning over segments with a notion of *amortization segment non-interference*, which along with some basic restrictions motivates the approach we describe here.

In Fig. 3, we show the control-flow graph of the resource-decomposed program in Fig. 2b without the inserted `reset`s. Node labels correspond to program points

there, except for labels 2^*, 4^*, 13^* that correspond to unlabeled program points in the initialization of the `for`-loops and the procedure exit. Edges are labeled by a single or a sequence of commands (where we omit keyword `assume` for brevity in the figure). Some nodes and edges are elided as ... that are not relevant for this discussion. Ignore node colors and the labels below the nodes for now.

Let us consider the class of *syntactic* selective-amortization transformations where we can rewrite resource use commands `use` r e to place `use`s into separate amortization groups, and we can insert a `reset` r' at a single program location to partition `use`s into amortization segments for each group r'. But otherwise, we make no other program transformation. We then use the notion of segment non-interference to select a group and segment decomposition under this syntactic restriction.

Now, the intuition behind amortization segment non-interference is that two segments for a resource r are non-interfering if under the same "low inputs," the resource usage of r is the same in both segments. In Fig. 3, the labels below the nodes show such low inputs to a particular `use` site from a particular program point. For example, under node 4, we show `p` as a low input for both the `use` sites at program points 6 and 11 (ignore the `:0`s for the moment).

So, an additional parameter in our search space is a partitioning of variables into "low" and "high" ones (which we note are not distinguished based on security relevance in the standard use of the non-interference term [1] but rather on relevance for amortized reasoning). We further fix the low variables in any segmentation we might use to be the internal variables on which the `use`s data-depend. This is based on the intuition that `use`s that share computation over internal variables are related for amortization. Because the `use`s for `#sb1` at program points 6 and 11 share `p` as an input at, for example, node 4, we place these `use` sites in the same group. Then, otherwise the other `use` sites at program points 8 and 12 are placed in other groups (namely, `#sb2` and `#sb3`, respectively). The set of variables on which `use` sites data-depend can be computed by a standard program slicing [37].

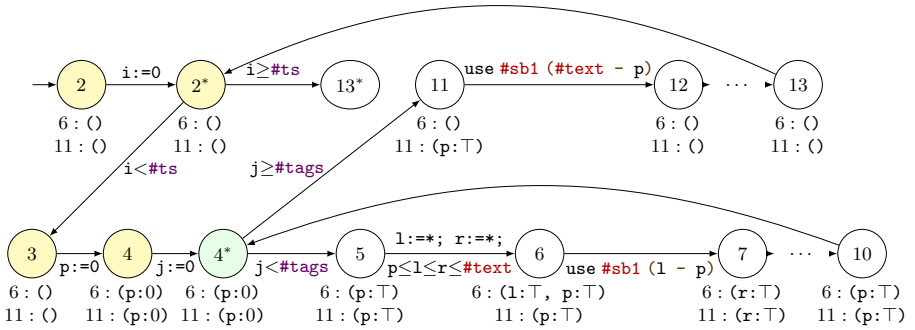

Fig. 3: Inserting a `reset` `#sb1` to select a segmentation for amortization group `#sb1`. We show the program from Fig. 2b here as a control-flow graph.

Finally, we insert a single `reset` for each group to define amortization segments. So that all `use` r e commands for a group r are always after some `reset` r, we consider program locations that control-dominate all `use` sites for r. In Fig. 3, any of the colored nodes control-dominate the two `use` sites for `#sb1`. To make the amortization segments as small as possible (while minimizing precision loss), we select the most immediate dominator where the low variables can be proven constant (i.e., the low inputs to the segments will always the same value). Node 4^* (colored green) is this dominator for the two `use` sites for `#sb1` because `p` is always 0 (shown as `p:0`) and where we insert `reset #sb1`. We can derive this constancy property with any numerical abstract domain (here, we show $\top$ for non-constant values from a standard constant propagation analysis for presentation), and we can pessimistically assume other variables to be low and also try to prove constancy for them to potentially recover some additional precision in segmentation.

Note that the analyses being applied here are classical ones. What is interesting here is not the analyses per se but their application to selecting amortization groups and segments to realize selectively-amortized resource bounding.

3 Decomposing Resource Usage

Our technique considers a resource-usage tracking program and splits a single resource variable into an arbitrary number of *resource decompositions*. By design, resource-usage tracking updates are generic in allowing updates with any integer-valued expression, enabling modeling non-monotonic resources like list additions and removals or memory allocation and deallocation. In this section, we define a core imperative language for resource-usage tracking, formalize selective-amortized analysis as a program transformation that inserts *amortization resets* into decomposed resource-usage tracking variables (Section 3.1), and show that any transformation is sound with respect to bound checks on resource usage (Section 3.2). While we focus on upper-bound checks, we will see that the approach can be easily adapted for lower-bound assertions.

In Fig. 4, we give the core resource-usage tracking language. We consider an unspecified expression language e, aside from including program variables x and its value forms v having integers n and booleans b. The command forms include standard imperative ones like the no-op unit `skip`, assignment x `:=` e, and guard condition `assume` e. The remaining highlighted command forms work with resources r. In particular, `use` r e models a resource use where the usage of r is incremented by the value of e, and `ub` $\overline{r}$ e is an upper-bound assertion checking that the sum of the resources $\overline{r}$ is upper-bounded by the value of e. We abuse notation slightly by writing $\overline{r}$ both for a sequence $r_1 \ldots r_n$ or a set $\{r_1, \ldots, r_n\}$ of resources. Selective amortization is realized through resetting resources with the `reset` r command that we detail further below. Note that program expressions e do not contain resources variables r. Finally, programs p are given as control-flow graphs with edges $\ell \dashv[c]\!\!\rightarrow \ell'$ labeled by commands c between locations ℓ.

values $v ::= n \mid b \mid \cdots$ booleans $b ::= \texttt{true} \mid \texttt{false}$ expressions $e ::= x \mid v \mid \cdots$

commands $c ::= \texttt{skip} \mid x \texttt{ := } e \mid \texttt{assume } e \mid \texttt{use } r\ e \mid \texttt{ub } \overline{r}\ e \mid \texttt{reset } r$
programs $p ::= \cdot \mid p, \ell \dashv[c]\vdash \ell'$

variables x resources r locations ℓ

stores $\rho ::= \cdot \mid \rho[x \mapsto v] \mid \rho[r \mapsto n] \mid \rho[r^* \mapsto n] \mid \rho[r^\sharp \mapsto n]$

$$\boxed{\langle \rho, e\rangle \Downarrow v \qquad \langle \rho, c\rangle \Downarrow \rho'}$$

E-Use
$$\frac{\langle \rho, e\rangle \Downarrow n}{\langle \rho, \texttt{use } r\ e\rangle \Downarrow \rho[r \mapsto \rho(r) + n]}$$

E-UBCheck
$$\frac{\langle \rho, e\rangle \Downarrow n \qquad \left(\sum_{r \in \overline{r}} \rho(r^\sharp) \cdot \rho(r^*) + \rho(r)\right) \leq n}{\langle \rho, \texttt{ub } \overline{r}\ e\rangle \Downarrow \rho}$$

E-Reset
$$\frac{\rho' = \rho[r^\sharp \mapsto \rho(r^\sharp) + 1][r^* \mapsto \max(\rho(r^*), \rho(r))][r \mapsto 0]}{\langle \rho, \texttt{reset } r\rangle \Downarrow \rho'}$$

Fig. 4: A core imperative language for resource-usage analysis. Resources r are modeled as integer-valued variables that may increase or decrease (via a `use` command) and bound-checked (via an `ub` assertion command). Selective amortization is realized through *resource* `reset`*s*.

The states σ of a program are pairs $\langle \ell : \rho\rangle$ of locations ℓ and stores ρ. Stores are finite maps, mapping program variables to values $x \mapsto v$, as well as tracking resources in the remaining highlighted forms. A resource r is a integer-valued variable $r \mapsto n$. For any resource r, we consider two auxiliary resource-usage summary variables r^* and $r^\sharp$ used in resource resetting to be described later.

A judgment form for evaluating expressions $\langle \rho, e\rangle \Downarrow v$ stands for "In store ρ, expression e evaluates to value v." Similarly, a judgment form $\langle \rho, c\rangle \Downarrow \rho'$ stands for "In store ρ, command c updates the store to ρ'." In Fig. 4, we elide the standard rules for `skip`, assignment x `:=` e, and guard condition `assume` e and focus on the resource-manipulating commands.

The E-Use rule captures that the `use` r e command says to increment r by the value of e. Note that we write $\rho(r)$ for looking up the mapping of r in store ρ and assume that any unmapped r maps to 0. That is, we consider all resources r initialized to 0. The E-UBCheck describes an upper-bound check `ub` $\overline{r}$ e on a set of resources $\overline{r}$. Let us first consider a single resource r and assume that the auxiliary variable r^* is 0 in the store. Then, the rule simply checks that r is upper-bounded by the value of e (i.e., like `assert` $r \leq e$). In the next subsection, we come back to the more general form of the upper-bound check shown in E-UBCheck, which captures the essence of selectively-amortized resource bounding through an interaction with resource decomposition and amortization resets.

$$\text{decompositions} \quad D ::= \cdot \mid D, r \multimap \overline{r}$$

$$\boxed{D \vdash c \multimap c'}$$

$$\text{D-Use}\ \frac{r' \in \overline{r}}{D, r \multimap \overline{r} \vdash \texttt{use}\ r\ e \multimap \texttt{use}\ r'\ e} \qquad \text{D-UBCheck}\ \frac{}{D, r \multimap \overline{r} \vdash \texttt{ub}\ r\ e \multimap \texttt{ub}\ \overline{r}\ e} \qquad \text{D-Reset}\ \frac{}{D \vdash \texttt{skip} \multimap \texttt{reset}\ r}$$

$$\text{D-Command}\ \frac{c \in \{\texttt{skip}, x\ \texttt{:=}\ e, \texttt{assume}\ e\}}{D \vdash c \multimap c}$$

Fig. 5: Decomposing resource usage for selective-amortization analysis is described with a transformation that rewrites commands with a resource decomposition D. Decompositions D define the amortization groups, while inserted `reset`s determine the amortization segments.

3.1 Selective Amortization By Decomposition

Recall from Section 2 that the essence of selectively-amortized resource bounding is we want to selectively choose the sequence of resource uses `use` r e over which we apply amortized reasoning. To do this, we have two intertwined tools: resource decomposition $r \multimap \overline{r}$ into *amortization groups* and amortization resets `reset` r into *amortization segments*.

A resource decomposition $D ::= \cdot \mid D, r \multimap \overline{r}$ is a mapping from a resource r into a set of decomposed resource-usage tracking variables $\overline{r}$. The transformation takes `use` r e and rewrites them to use `use` r' e for some $r' \in \overline{r}$, thus decomposing all uses of r into separate amortization groups given by $\overline{r}$. In Fig. 5, the judgment form $D \vdash c \multimap c'$ says, "Under resource decomposition D, command c can be resource-decomposed to command c'," stating valid decomposition transformations. The D-Use rule states exactly this transformation for `use` r e commands.

Then, within separate amortization groups, resets `reset` r define the *segments* of execution over which to amortize resource uses while applying worst-case reasoning around them. To see this, consider the E-Reset rule in Fig. 4 where we can see `reset` r as corresponding to the following assignments (abusing notation slightly with assignments and expressions using resource variables):

$$r^\sharp\ \texttt{:=}\ r^\sharp + 1; \quad r^*\ \texttt{:=}\ \max(r^*, r); \quad r\ \texttt{:=}\ 0;$$

That is, the `reset` r command increments the number of amortization segments for r seen so far in $r^\sharp$, saves the maximum value of r in any segment so far in r^*, and resets r to 0 ending the last amortization segment and starting the next one. So the r^* resource-usage summary captures the worst-case resource use of r over all segments, while the $r^\sharp$ summary saves the number of such amortization segments.

These summaries then enables amortized reasoning within segments and worst-case reasoning around them. To see this, let us consider a one-to-one resource decomposition $r_\circ \multimap r$. Without loss of generality, we assume the original

program using r_o does not have any `reset`s (but the transformed program with r may). Furthermore, we assume all amortization segments are paths of the form $\rho\,\texttt{reset}\ r \cdots \rho'\,\texttt{reset}\ r$ with no other `reset` r in the middle and that there are no resource uses `use` $r\ e$ before an initial `reset` r (i.e., all executions of `use` $r\ e$ are either in a segment bracketed by two `reset` rs or after the last `reset` r). Then the following selective-amortization assertion between r_o and r holds globally (in all reachable stores):

$$r_o \quad \leq \quad r^\sharp \cdot r^* + r$$

Intuitively, up to the last `reset` r, there have been $r^\sharp$ amortization segments and the worst-case use of r on all prior segments is r^*, so $r^\sharp \cdot r^*$ is an upper bound on the resource use up to the the last `reset` r—thereby using worst-case reasoning on amortized segments. Then we just add r because the remaining uses `use` $r\ e$ since the last reset have accumulated in r. Note that we thus consider all upper-bound summaries r^* initialized to 0 and all segment-counter summaries $r^\sharp$ initialized to -1.

Coming back to the E-UBCheck rule describing the upper-bound check `ub` $r\ e$ in Fig. 4 (for a single resource r), the assertion checks the bound e on exactly this amortized segments expression (i.e., like `assert` $r^\sharp \cdot r^* + r \leq e$). Then, with respect to amortization groups, a resource decomposition $r \multimap \overline{r}$ says that resource uses to r are distributed over uses to $\overline{r}$, so we simply sum over the amortization groups $\overline{r}$ (i.e., like `assert` $\left(\sum_{r \in \overline{r}} r^\sharp \cdot r^* + r\right) \leq e$).

Thus, the transformation from an upper-bound check `ub` $r\ e$ on a resource r with decomposition $r \multimap \overline{r}$ yields `ub` $\overline{r}\ e$ as stated in rule D-UBCheck from Fig. 5. As alluded to above, it is sound to insert `reset`s arbitrarily into the transformed program corresponding to different amortization segments, which we state with rule D-Reset. Note that we consider programs p equivalent up to insertions of `skip` commands, so we can insert them into the original program as needed. The remaining non-resource manipulating commands are simply retained as-is with rule D-Command. For simplicity in presentation, we assume the original program does not have `reset`s and has only single-resource upper-bound checks `ub` $r\ e$. Overall, any choice of a resource decomposition D is sound corresponding to different amortization groups. Again for simplicity, we assume all resources r in the original program have a mapping in D (e.g., at least have $r \multimap r$ for no decomposition). We consider soundness in more detail further below.

3.2 Soundness of Group and Segment Decomposition

To consider the soundness of the resource decomposition transformation $D \vdash c \multimap c'$, we define program executions or paths π. In Fig. 6, we define paths π in a slightly non-standard way: they are sequences created by appending a state $\pi\,\sigma$ or appending a store-command pair $\pi\,\rho\,c$ and are well-formed if they consist of sequences corresponding to the stores from valid executions of the commands (as captured by the π `ok` judgment). Intentionally, we define paths mostly independent from programs, stripping out locations ℓ except for the last state $\langle \ell : \rho \rangle$. In most cases, we do not care about the program from which paths

$$\text{states } \sigma ::= \langle \ell \colon \rho \rangle \qquad \text{paths } \pi \in \Pi ::= \cdot \mid \pi\, \sigma \mid \pi\, \rho\, c$$

$$\boxed{\pi\ \mathsf{ok} \qquad \sigma \to_p \sigma' \qquad [\![p]\!]\sigma = \Pi}$$

$$\textsc{Ok-Init}\ \frac{}{\sigma\ \mathsf{ok}} \qquad \textsc{Ok-Step}\ \frac{\pi\, \langle \ell \colon \rho \rangle\ \mathsf{ok} \qquad \langle \rho, c \rangle \Downarrow \rho'}{\pi\, \rho\, c\, \langle \ell' \colon \rho' \rangle\ \mathsf{ok}} \qquad \textsc{Step}\ \frac{\ell \dashv [c] \mapsto \ell' \in p \qquad \langle \rho, c \rangle \Downarrow \rho'}{\langle \ell \colon \rho \rangle \to_p \langle \ell' \colon \rho' \rangle}$$

$$[\![p]\!]\sigma \stackrel{\text{def}}{=} \mathrm{lfp}\, \lambda \Pi.\, \{\sigma\} \cup \bigcup_{\pi\, \langle \ell \colon \rho \rangle \in \Pi} \left\{\, \pi\, \rho\, c\, \sigma' \;\middle|\; \langle \ell \colon \rho \rangle \to_p \sigma' \,\right\}$$

$$\boxed{D \vdash \pi \mathrel{-\!\!\in} \pi'}$$

$$\textsc{D-AppendCommand}\ \frac{D \vdash \pi\, \langle \ell \colon \rho \rangle \mathrel{-\!\!\in} \pi'\, \langle \ell' \colon \rho' \rangle \quad D \vdash c \mathrel{-\!\!\in} c'}{D \vdash \pi\, \rho\, c \mathrel{-\!\!\in} \pi'\, \rho'\, c'} \qquad \textsc{D-Step}\ \frac{D \vdash \pi \mathrel{-\!\!\in} \pi' \quad \pi'\, \sigma'\ \mathsf{ok} \quad \sigma \leqq_D \sigma'}{D \vdash \pi\, \sigma \mathrel{-\!\!\in} \pi'\, \sigma'} \qquad \textsc{D-Init}\ \frac{}{D \vdash \sigma \mathrel{-\!\!\in} \sigma}$$

$$\boxed{\rho \leqq_D \rho' \qquad \sigma \leqq_D \sigma'}$$

$$\rho_{\mathrm{o}} \leqq_D \rho \ \text{ iff } \ \rho_{\mathrm{o}}(x) = \rho(x) \text{ for all } x \in \mathrm{vars}(\rho_{\mathrm{o}}) = \mathrm{vars}(\rho) \text{ and}$$
$$\rho_{\mathrm{o}}(r_{\mathrm{o}}) \leq \sum_{r \in D(r_{\mathrm{o}})} \rho(r^\sharp) \cdot \rho(r^*) + \rho(r) \text{ for all } r_{\mathrm{o}} \in \mathrm{dom}(\rho_{\mathrm{o}})$$

$$\langle \ell \colon \rho \rangle \leqq_D \langle \ell' \colon \rho' \rangle \ \text{ iff } \ \rho \leqq_D \rho' \qquad \mathrm{vars}(\rho) \stackrel{\text{def}}{=} \{\, x \mid x \in \mathrm{dom}(\rho) \,\}$$

$$\boxed{D \vdash p \mathrel{-\!\!\in} p'}$$

$$\textsc{D-Transition}\ \frac{D \vdash c \mathrel{-\!\!\in} c'}{D \vdash p, \ell \dashv [c] \mapsto \ell' \mathrel{-\!\!\in} p', \ell \dashv [c'] \mapsto \ell'} \qquad \textsc{D-EmptyProgram}\ \frac{}{D \vdash \cdot \mathrel{-\!\!\in} \cdot}$$

Fig. 6: A semantic decomposition is captured with a path transformation $D \vdash \pi \mathrel{-\!\!\in} \pi'$ where paths π are sequences of command executions. The path transformation says we can rewrite according to the command transformation until reaching the same initial state. That is, choosing amortization groups with any decomposition D and amortization segments with any insertions of `reset`s are sound. A syntactic decomposition is simply a lifting of the command transformation to programs $D \vdash p \mathrel{-\!\!\in} p'$ on the same control-flow structure.

may come from. For example, the path well-formedness judgment $\pi\ \mathsf{ok}$ ignores program locations and simply checks that the triples of store ρ, command c, and store ρ' are valid executions $\langle \rho, c \rangle \Downarrow \rho'$ (rule Ok-Step). Unless otherwise stated, we assume all paths π are well formed (i.e., $\pi\ \mathsf{ok}$ holds for any path π).

The only reason paths mention locations is to define the path semantics $[\![p]\!]\sigma$ of a program p with initial state σ. The path semantics $[\![p]\!]\sigma$ is given as: (1) the judgment form $\sigma \to_p \sigma'$ defines a transition relation saying, "On program p,

state σ steps to state σ'," and (2) the path semantics $[\![p]\!]\sigma$ collects all finite (but unbounded) prefixes of the transition system from the initial state σ.

The judgment form $D \vdash \pi \multimap \pi'$ states a selectively-amortized resource bounding on a path π' from an original path π. Divorcing paths from programs emphasizes that semantically, we can choose any amortization grouping with a choice of the resource decomposition D and select any amortization segmentation by inserting `reset`s anywhere along the original path π. The D-AppendCommand rules says that a command along the original path can be rewritten according to the command transformation $D \vdash c \multimap c'$. Note that like with programs, we consider paths π equivalent up to insertions of `skip` commands, so we can insert them into the original path as needed.

To talk about resource-decomposed stores along paths, we define $\rho \leqq_D \rho'$ to be stores that are equal on program variables $\mathrm{vars}(\rho)$ (excluding resource variables r) and whose resource-usage tracking variables satisfy the selectively-amortized assertion from Section 3.1 (see Fig. 6 for a detailed definition). Then, the D-Step rule says that the execution of the last command in π' must result in a state σ' consistent with the semantics of commands ($\pi'\,\sigma'\,\mathsf{ok}$) and with selective amortization ($\sigma \leqq_D \sigma'$). Finally, the D-Init rule simply says that resource-decomposed paths should start with the same initial state.

We can then consider a more restricted, syntactic class of selectively-amortized resource-bounding transformations by simply transforming the commands of a program p (i.e., the judgment form $D \vdash p \multimap p'$ in Fig. 6). To achieve more semantic selective amortizations, one could, of course, first apply richer semantics-preserving program transformations to the original program (than inserting `skip`s) before applying the resource-decomposition transformation.

We can now state the following soundness result.

Theorem 1 (Soundness of Selectively-Amortized Resource Bounding).

1. *If* $D \vdash c_\mathrm{o} \multimap c$, $\langle \rho, c\rangle \Downarrow \rho'$, *and* $\rho_\mathrm{o} \leqq_D \rho$, *then* $\langle \rho_\mathrm{o}, c_\mathrm{o}\rangle \Downarrow \rho'_\mathrm{o}$ *with* $\rho'_\mathrm{o} \leqq_D \rho'$.
2. *If* $D \vdash \pi_\mathrm{o} \multimap \pi$ *and* $\pi\ \mathsf{ok}$, *then* $\pi_\mathrm{o}\ \mathsf{ok}$.
3. *If* $D \vdash p_\mathrm{o} \multimap p$ *and* $\pi \in [\![p]\!]\sigma$, *then there is a* $\pi_\mathrm{o} \in [\![p_\mathrm{o}]\!]\sigma$ *s.t.* $D \vdash \pi_\mathrm{o} \multimap \pi$.

The key lemma (part 1) states a preservation property that any command decomposition preserves the selectively-amortized resource-bounding invariant $\leqq_D$ (see the extended version [29] for details).

Verifying Bounds with Selective Amortization. Bound verification by selective amortization follows directly from the soundness theorem given above. In particular, given a particular resource composition D and a transformed program p from the original program p_o such that $D \vdash p_\mathrm{o} \multimap p$, simply apply any off-the-shelf numerical verification or invariant generator to p to try to prove translated upper-bound assertions $\mathtt{ub}\ \overline{r}\ e$ in p.

In Section 4, we describe an approach for selecting a resource decomposition and inserting amortization resets. However, we note that our key contribution described here is generically defining the space of selective amortizations.

Lower Bounds. While we focused on upper-bound checks in this section, we see that the approach can be adapted to lower-bound assertions in a straightforward manner by introducing a lower-bound resource-usage summary variable, say $r^\dagger$. This lower-bound summary is analogously updated on `reset` r with the minimum resource-usage so far (i.e., like $r^\dagger$ `:=` $\min(r^\dagger, r)$). We can then translate lower-bound assertions `lb` e r in the analogous manner and extend the selectively-amortized resource bounding invariant $\leqq_D$ for lower bounds.

4 Selecting a Decomposition

In this section, we describe a way to select amortization groups (i.e., a resource decomposition D) and amortization segments (i.e., insertions of amortization `reset`s) to algorithmically realize selectively-amortized resource bounding. As alluded to in Section 2, there is a tension between creating as many groups and as short segments as possible to focus amortized reasoning only where it is needed, simplifying the invariant inference needed to do so, versus not creating too many groups or too short segments that the needed amortization for precision is lost. More specifically, the built-in multiplication $r^\sharp \cdot r^*$ we apply for worst-case reasoning around segments simplifies the necessary invariants needed to prove bounds but only if r^* is sufficiently precise bound on resource usage per segment.

As hinted at in Section 3, the space of possible selective amortizations is huge. Even with some basic restrictions to make this search more feasible, the remaining space of selective amortizations is still large. In the remainder of this section, we first characterize when the resource-usage summary r^* is precise based on a notion of *non-interfering amortization segments*. Then, we describe the basic restrictions and their motivations to use segment non-interference to search within this restricted space.

Non-Interfering Amortization Segments. Recall the selective-amortization assertion $r_\mathrm{o} \leq r^\sharp \cdot r^* + r$ and the r^* `:=` $\max(r^*, r)$ update for a `reset` r from Section 3.1. We can see that the difference between the sides of the inequality (i.e., $(r^\sharp \cdot r^* + r) - r_\mathrm{o}$) comes from a difference between the current upper-bound summary r^* and the current resource accumulation in r (i.e., $r^* - r$) on a `reset` r. Thus intuitively, we want to insert amortization resets `reset` r at locations that would minimize this difference $r^* - r$ across all such amortization segments. This observation suggests a definition for segment non-interference:

Definition 1 (Amortization Segment Non-Interference). Consider two paths $\pi\colon (\rho_\mathrm{lo} \uplus \rho_\mathrm{hi})$ `reset` $r \cdots \rho$ `reset` r and $\pi'\colon (\rho_\mathrm{lo} \uplus \rho'_\mathrm{hi})$ `reset` $r \cdots \rho'$ `reset` r such that $\mathrm{dom}(\rho_\mathrm{hi}) = \mathrm{dom}(\rho'_\mathrm{hi})$. That is, we consider two amortization segments (i.e., paths that start and end in a `reset` r) and partition the input into low variables (i.e., $\mathrm{dom}(\rho_\mathrm{lo})$) and high variables (i.e., $\mathrm{dom}(\rho_\mathrm{hi})$). Then, we say segments π and π' are *non-interfering* iff for any (high) stores ρ_hi and ρ'_hi, and for any (low) store ρ_lo, we have that $\rho(r) = \rho'(r)$.

We see that if all pairs of amortization segments are non-interfering for a suitable partition of variables between high and low variables, then the selective amortization is as precise as the fully amortized solution. Then, we want to balance making amortization segments as small as possible (in order to simplify invariant inference and maximize worst-case reasoning) with the smallest set of low input variables (to maximize non-interference).

Computed Input-Independent Groups and Single Location-Based Segments. Definition 1 suggests an approach to selecting amortization groups and segments if we fix some basic restrictions: (1) First, we consider syntactic decomposition transformations $D \vdash p_{\text{o}} \dashv\in p$ from the original program p_{o}. (2) Second, we consider a single insertion of `reset` r into the transformed program p that control-dominates all uses `use` r e for every resource r. Picking a control-dominating location ℓ ensures we do not have any `use` r e before a `reset` r, and performing single insertion means we only need to consider segments that start and end at *single location* ℓ (where $\ell \dashv[\texttt{reset}\ r]\mapsto \ell' \in p$). (3) Third, we fix the low variables in any segmentation we consider to be the internal variables on which the `use`s data-depends, leaving any remaining variables at the segment start location ℓ to be high, including the inputs to the entry location of the original program p_{o}. Intuitively, we assume that `use`s that share computation over internal, low variables are related for amortization. However, there is still significant flexibility in choosing the resource decomposition D that defines the amortization groups and the `use`s-dominating location ℓ for each resource r in the transformed program p—it does not have to be the immediate dominator of the `use`s.

As we want to create more groups to simplify invariant inference, let us first consider the resource decomposition D such that each syntactic `use` r e in p_{o} is translated to a unique resource variable and thus placed in a distinct group (i.e., such that $|D(r)| = |\{ (\ell, e, \ell') \mid \ell \dashv[\texttt{use}\ r\ e]\mapsto \ell' \in p_{\text{o}} \}|$). However, to find cases where distinct groups are potentially insufficient, we consider possibly merging `use` sites pairwise (i.e., $\ell_1 \dashv[\texttt{use}\ r_1\ e_1]\mapsto \ell_1'$ and $\ell_2 \dashv[\texttt{use}\ r_2\ e_2]\mapsto \ell_2'$ in the transformed program p). Suppose we were to merge groups r_1 and r_2, then let us consider the immediate dominator ℓ of locations ℓ_1 and ℓ_2, which defines the possible amortization segments starting from and ending at location ℓ. Considering this potential segmentation and the shared low input variables that may affect the value of both r_1 and r_2 and if the values of these low input variables may change in the segment, then we want to merge these groups based on restriction (3) above (otherwise, they are *computed input independent*). We can then approximate this criteria with standard, backwards data-dependency slices [37] from the uses `use` r_1 e_1 and `use` r_2 e_2.

Once we have fixed a resource decomposition D defining amortization groups, selecting a location ℓ to insert each `reset` r for each r in the transformed program p is fairly straightforward. Following segment non-interference, for any `use` sites sharing the same resource r (i.e., $L = \{ \ell \mid \ell \dashv[\texttt{use}\ r\ e]\mapsto \ell' \in p \}$), find the most immediate dominator of L where we can prove that the low input variables are constant (i.e., call this `use`-dominating location ℓ, then we have that $\rho(x_{\text{lo}}) = n$ for some n, for all low input variables x_{lo}, in all reachable states $\langle \ell \colon \rho \rangle$). If we

can prove that the low input variables are constant in the program up to the amortization segment entry location ℓ, then we satisfy segment non-interference (up to non-determinism within segments).

Note that because of the tension between precision from simplifying invariant inference versus from amortization, selecting a decomposition is necessarily heuristic. Section 3 shows that picking any decomposition is sound, and Sect. 5 offers evidence that the principled heuristic described here provides a benefit.

5 Empirical Evaluation

Selective amortization represents a large space of possible approaches between worst-case and fully amortized reasoning. Here we attempt to provide evidence that selective amortization provides a benefit when compared with the two extremes, even with simply the heuristic decomposition strategy described in Sect. 4. It is this specific selective amortization strategy that we consider here in our experiments. We consider the following research question on *Effectiveness*: Can selective amortization improve the number of verified programs when compared with the worst-case and fully-amortized extremes?

Effectiveness. In Table 1, we summarize the comparison between selective amortization and the two extremes with the most precise configuration in each category **bolded**. For each category, we list the number of programs (num) and the total lines of code (loc). To test the effect of slightly weaker bound assertions, we consider two sets of assertions: for the most precise bounds and by relaxing the constant coefficients from the most precise bounds. For each configuration, we applied the same verification tools after transformation with our tool Brbo [30] implemented in 6,000 lines of Scala, using Z3 [14] for SMT solving and ICRA [27] as an off-the-shelf invariant generator. For the two sets of

Table 1: Verifying with worst-case (Wor), fully-amortized (Ful), and selectively-amortized (Sel) with two sets of assertions: the most precise bounds and constant-weakened ones. For each configuration, we give the number of assertions proven (n) and the total verification time in seconds (s).

			Most Precise Bounds						Constant-Weakened Bounds					
			Wor		Ful		Sel		Wor		Ful		Sel	
category	num	loc	(n)	(s)	(n)	(s)	(n)	(s)	(n)	(s)	(n)	(s)	(n)	(s)
lang3	20	667	**12**	175.7	8	44.2	**12**	249.5	**14**	302.7	**14**	89.0	**14**	252.0
stringutils	10	390	2	12.9	**4**	196.5	**4**	176.2	4	101.3	5	209.0	**6**	264.3
guava	3	90	0	0	**1**	7.6	0	0	2	18.3	**3**	30.0	**3**	73.3
stac	3	122	2	118.6	2	23	**3**	101.1	2	126.6	2	22.5	**3**	105.2
generated	200	3633	139	1510.2	43	198.3	**175**	1779.5	140	1567.8	69	325.3	**180**	1852.2
total	236	4902	155	1817.4	58	469.6	**194**	2306.3	162	2116.7	93	675.8	**206**	2547.0

bound assertions, 194 and 206 programs, respectively, were verified for the selective amortization configuration—more than the number with either extreme. The improvement over worst-case reasoning comes from amortizing the costs over multiple commands, while the improvement over fully-amortized reasoning comes from amortizing the costs over subprograms that are smaller than the whole program, so that inferring invariants becomes more manageable for ICRA.

The verification time in Table 1 consists of selecting amortizations, realizing amortizations via program transformations, and verifying bound assertions on the transformed programs, which include invariant generation. We observed that selecting amortizations and realizing them via program transformations consumed negligible amounts of time; invariant generation took up more than 95% of the total time. Selecting amortizations based on the approach described in Sect. 4 is fast because the selection only requires simple data- and control-dependency analysis.

As noted above, these experiments consider the specific decomposition strategy described in Sect. 4 *on the original benchmarks*, even though we show in Sect. 3 that picking any decomposition is sound. But as alluded to in Sect. 3, programs can be transformed in semantics-preserving ways that then expose different possible decompositions (to either the strategy described in Sect. 4 or even some other one). Others have made similar observations; for example, semantic program transformations that split a loop into multiple phases [32] may simplify the invariant generation by reducing the need for disjunctive invariants. Indeed, it may strike the best balance between scalability and precision if we can effectively perform different semantic transformations based on the precision we need for proving some desired bounds.

Benchmarks. We developed this benchmark suite specifically for the resource bounding problem (as it differs from, for example, the loop bounding problem). In particular, we collected code from 36 real-world programs (from 4 libraries or suites) that use `StringBuilder`. Furthermore, we created a suite of 200 synthetic programs generated by randomly nesting and sequencing two common loop idioms that are extracted from actual Java programs.

6 Related Work

Loop Bound Analysis and Worst-Case Reasoning. A large body of work has addressed bounding the number of loop iterations in imperative numeric programs [8, 10, 11, 19–21, 33, 35, 39]. These techniques rely on ranking functions to quantitatively track the changes in the rankings of states. The loop bounding problem can be seen as a special case of the resource bounding problem where the resource of interest is loop iteration, and the cost of each "use" (i.e., iteration) is a constant 1. Or in other words, the loop bounding problem can be extended to address the resource bounding problem, if one adopts what we call worst-case reasoning to fix a constant upper bound for each resource use. There are works that essentially take this perspective to apply loop bound analysis for invariant inference [8, 9, 35].

Worst-case execution time (WCET) analysis [38] is an area of study that attempts to automatically infer time bounds for machine code, considering precise models of hardware architectures. It can be seen as another instance of worst-case reasoning, focusing on defining precise worst-case bound models for instructions but generally assuming loop bounds are given or easy to derive.

Our approach is partly inspired by Gulwani et al. [21] that describes a loop bound analysis because we also rely on a program transformation to simplify the forms of the needed inductive invariants. At the same time, we improve on this work by first generalizing the reasoning of loop iterations to general resources, which can change in a non-trivial (i.e., non-monotonic and non-constant) way, and then introduce *selective amortization* that mixes in amortized reasoning from the next category of papers to address these challenges.

(Fully-)Amortized Reasoning. Several lines of work employ a number of different techniques to precisely reason about resource usage over *full* executions (i.e., attempt to perform fully-amortized reasoning). The COSTA project [2–6], which adopts the recurrence relation approach, reasons about resource usage by first abstracting program semantics into a set of recurrence relations and then finding closed-form solutions to these recurrence relations. The RAML project [22, 23] analyzes the resource usage of functional programs with the potential method. This approach encodes the changes of a potential with linear programming constraints over the unknown coefficients of pre-determined bound templates. Carbonneaux et al. [10, 11] adapts this approach to numerical imperative programs. Atkey [7] (and improvements [17, 18]) develop expressive program logics that extend type-based amortized resource analysis with resource reasoning over heap data structures. The above approaches can be viewed as instances of fully-amortized reasoning, because it is in an amortized manner that they encode the sum of the resource usage into systems of constraints [2–6, 10, 11] or perform deductive proofs that amortize costs [7, 17, 18]. The key challenge in fully-amortized reasoning is to infer complex inductive invariants, which are the solutions of the constraint systems in, for example, COSTA and RAML. Instead, our approach may simplify the forms of the required invariants by decomposing the resource usage into groups and segments of amortized costs. Since our approach is agnostic to the underlying amortized reasoning engine, any fully-amortized reasoning approach, such as the above ones, can potentially be used in place of the relational inductive invariant generator applied in this paper.

7 Conclusion

In this paper we address the problem of automatically proving resource bounds, where resource usage is expressed via an integer-typed variable. We present a framework for selectively-amortized reasoning that mixes worst-case and fully amortized reasoning via a property decomposition and a program transformation. We show that proving bounds in any such decomposition yields a sound resource bound in the original program, and we give an algorithm for selecting an

effective decomposition. Our empirical evaluation provides evidence that selective amortization effectively leverages both worst-case and amortized reasoning.

Acknowledgements We thank Pavol Černý for his valuable contributions in the early stages of this research. We also thank the anonymous reviewers and members of the CUPLV lab for their helpful reviews and suggestions. This research was supported in part by the Defense Advanced Research Projects Agency under grant FA8750-15-2-0096, and also by the National Science Foundation under grant CCF-2008369.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Principles of Programming Languages (POPL)*, pages 147–160, 1999. URL https://doi.org/10.1145/292540.292555.
2. Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In *European Symposium on Programming (ESOP)*, volume 4421, pages 157–172, 2007. URL https://doi.org/10.1007/978-3-540-71316-6_12.
3. Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Heap space analysis for Java bytecode. In *International Symposium on Memory Management (ISMM)*, pages 105–116, 2007. URL https://doi.org/10.1145/1296907.1296922.
4. Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *International Symposium on Memory Management (ISMM)*, pages 129–138, 2009. URL https://doi.org/10.1145/1542431.1542450.
5. Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538, pages 38–53, 2011. URL https://doi.org/10.1007/978-3-642-18275-4_5.
6. Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In *Static Analysis (SAS)*, volume 7460, pages 405–421, 2012. URL https://doi.org/10.1007/978-3-642-33125-1_27.
7. Robert Atkey. Amortised resource analysis with separation logic. In *European Symposium on Programming (ESOP)*, volume 6012, pages 85–103, 2010. URL https://doi.org/10.1007/978-3-642-11957-6_6.
8. Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Alternating runtime and size complexity analysis of integer programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413, pages 140–155, 2014. URL https://doi.org/10.1007/978-3-642-54862-8_10.
9. Pavel Cadek, Clemens Danninger, Moritz Sinn, and Florian Zuleger. Using loop bound analysis for invariant generation. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018. URL https://doi.org/10.23919/FMCAD.2018.8603005.
10. Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Programming Language Design and Implementation (PLDI)*, pages 467–478, 2015. URL https://doi.org/10.1145/2737924.2737955.

11. Quentin Carbonneaux, Jan Hoffmann, Thomas W. Reps, and Zhong Shao. Automated resource analysis with Coq proof objects. In *Computer-Aided Verification (CAV)*, volume 10427, pages 64–85, 2017. URL https://doi.org/10.1007/978-3-319-63390-9_4.
12. Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. Polynomial invariant generation for non-deterministic recursive programs. In *Programming Language Design and Implementation (PLDI)*, pages 672–687, 2020. URL https://doi.org/10.1145/3385412.3385969.
13. Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *Computer-Aided Verification (CAV)*, volume 2725, pages 420–432, 2003. URL https://doi.org/10.1007/978-3-540-45069-6_39.
14. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963, pages 337–340, 2008. URL https://doi.org/10.1007/978-3-540-78800-3_24.
15. Defense Advanced Research Projects Agency (DARPA). Space/time analysis for cybersecurity (STAC), 2019. URL https://www.darpa.mil/program/space-time-analysis-for-cybersecurity.
16. Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 443–456, 2013. URL https://doi.org/10.1145/2509136.2509511.
17. Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming (ESOP)*, volume 10801, pages 533–560, 2018. URL https://doi.org/10.1007/978-3-319-89884-1_19.
18. Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal proof and analysis of an incremental cycle detection algorithm. In *Interactive Theorem Proving (ITP)*, volume 141, pages 18:1–18:20, 2019. URL https://doi.org/10.4230/LIPIcs.ITP.2019.18.
19. Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *Programming Language Design and Implementation (PLDI)*, pages 292–304, 2010. URL https://doi.org/10.1145/1806596.1806630.
20. Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Programming Language Design and Implementation (PLDI)*, pages 375–385, 2009. URL https://doi.org/10.1145/1542476.1542518.
21. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL)*, pages 127–139, 2009. URL https://doi.org/10.1145/1480881.1480898.
22. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Principles of Programming Languages (POPL)*, pages 357–370, 2011. URL https://doi.org/10.1145/1926385.1926427.
23. Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*, pages 359–373, 2017. URL https://doi.org/10.1145/3009837.3009842.
24. Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. Polynomial invariants for affine programs. In *Logic in Computer Science (LICS)*, pages 530–539, 2018. URL https://doi.org/10.1145/3209108.3209142.

25. Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. Compositional recurrence analysis revisited. In *Programming Language Design and Implementation (PLDI)*, pages 248–262, 2017. URL https://doi.org/10.1145/3062341.3062373.
26. Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.*, 2(POPL):54:1–54:33, 2018. URL https://doi.org/10.1145/3158142.
27. Zachary Kincaid, Jason Breck, John Cyphert, and Thomas W. Reps. Closed forms for numerical loops. *Proc. ACM Program. Lang.*, 3(POPL):55:1–55:29, 2019. URL https://doi.org/10.1145/3290368.
28. Tianhan Lu, Pavol Cerný, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Type-directed bounding of collections in reactive programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 11388, pages 275–296, 2019. URL https://doi.org/10.1007/978-3-030-11245-5_13.
29. Tianhan Lu, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Selectively-amortized resource bounding (extended version), 2021. URL https://arxiv.org/abs/2108.08263.
30. Tianhan Lu, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Selectively-amortized resource bounding (artifact), 2021. URL https://zenodo.org/record/5140586.
31. Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83(1):91–130, 1991. URL https://doi.org/10.1016/0304-3975(91)90041-Y.
32. Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Computer-Aided Verification (CAV)*, volume 6806, pages 703–719, 2011. URL https://doi.org/10.1007/978-3-642-22110-1_57.
33. Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *Computer-Aided Verification (CAV)*, volume 8559, pages 745–761, 2014. URL https://doi.org/10.1007/978-3-319-08867-9_50.
34. Moritz Sinn, Florian Zuleger, and Helmut Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 144–151, 2015.
35. Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reason.*, 59(1):3–45, 2017. URL https://doi.org/10.1007/s10817-016-9402-4.
36. Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.
37. Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984. URL https://doi.org/10.1109/TSE.1984.5010248.
38. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008. URL https://doi.org/10.1145/1347375.1347389.
39. Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *Static Analysis (SAS)*, volume 6887, pages 280–297, 2011. URL https://doi.org/10.1007/978-3-642-23702-7_22.