Non-cumulative Resource Analysis

Elvira Albert[1], Jesús Correas Fernández[1], and Guillermo Román-Díez[2]

[1] DSIC, Complutense University of Madrid, Spain
[2] DLSIIS, Technical University of Madrid, Spain

Abstract. Existing cost analysis frameworks have been defined for cumulative resources which keep on increasing along the computation. Traditional cumulative resources are execution time, number of executed steps, amount of memory allocated, and energy consumption. Non-cumulative resources are acquired and (possibly) released along the execution. Examples of non-cumulative cost are memory usage in the presence of garbage collection, number of connections established that are later closed, or resources requested to a virtual host which are released after using them. We present, to the best of our knowledge, the first generic static analysis framework to infer an *upper bound* on the *peak cost* for non-cumulative types of resources. Our analysis comprises several components: (1) a pre-analysis to infer when resources are being used simultaneously, (2) a *program-point* resource analysis which infers an upper bound on the cost at the points of interest (namely the points where resources are acquired) and (3) the elimination from the upper bounds obtained in (2) of those resources accumulated that are not used simultaneously. We report on a prototype implementation of our analysis that can be used on a simple imperative language.

1 Introduction

Cost analysis (a.k.a. resource analysis) aims at statically (without executing the program) inferring *upper bounds* on the resource consumption of the program as functions of the input data sizes. Traditional resources (e.g., time, steps, memory allocation, number of calls) are *cumulative*, i.e., they always increase along the execution. Ideally, a cost analysis framework is *generic* on the type of resource that the user wants to measure so that the resource of interest is a parameter of the analysis. Several generic cost analysis frameworks have been defined for cumulative resources using different formalisms. In particular, the classical framework based on recurrence relations has been used to define a cost analysis for a Java-like language [2]; approaches based on program invariants are defined in [11,14]; type systems have been presented in [15].

Non-cumulative resources are first acquired and then released. Typical examples are memory usage in the presence of garbage collection, maximum number of connections established simultaneously, the size of the stack of activation records, etc. The problem is nowadays also very relevant in *virtualized* systems, as in cloud computing, in which resources are acquired when needed and released after being used. It is recognized that non-cumulative resources introduce new

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 85–100, 2015.
DOI: 10.1007/978-3-662-46681-0_6

challenges in resource analysis [5,12]. This is because the resource consumption can increase and decrease along the computation, and it is not enough to reason on the final state of the execution, but rather the upper bound on the cost can happen at any intermediate step. We use the term *peak cost* to denote such maximum cost of the program execution for non-cumulative resources.

While the problem of inferring the peak cost has been studied in the context of memory usage for specific models of garbage collection [5,8,12], a generic framework to estimate the non-cumulative cost does not exist yet. The contribution of this paper is a generic resource analysis framework for a today's imperative language enriched with instructions to acquire and release resources. Thus, our framework can be instantiated to measure any type of non-cumulative resource that is acquired and (optionally) freed. The analysis is defined in two steps which are our main contributions: (1) We first infer the sets of resources which can be in use simultaneously (i.e., they have been both acquired and none of them released at some point of the execution). This process is formalized as a static analysis that (over-)approximates the sets of acquire instructions that can be in use simultaneously, allowing us to capture the simultaneous use of resources in the execution. (2) We then perform a *program-point* resource analysis which infers an upper bound on the cost at the points of interest, namely the points at which the resources are acquired. From such upper bounds, we can obtain the peak cost by just eliminating the cost due to acquire instructions that do not happen simultaneously with the others (according to the analysis information gathered at step 1). Additionally, we describe an extension of the framework which can improve the accuracy of the upper bounds by accounting only once the cost introduced at program points where resources are allocated and released repeatedly. Finally, we illustrate how the framework can be extended to get upper bounds for programs that allocate different kinds of resources.

We demonstrate the accuracy and feasibility of our approach by implementing a prototype analyzer for a simple imperative language. Preliminary experiments show that the non-cumulative resource analysis achieves gains up to 92.9% (on average 53.9%) in comparison to a cumulative resource analysis. The analysis can be used online from a web interface at `http://costa.ls.fi.upm.es/noncu`.

2 The Notion of Peak Cost

We start by defining the notion of peak cost that we aim at over-approximating by means of static analysis in the concrete setting.

2.1 The Language

The framework is developed on a language which is deliberately simple to define the analysis in a clear way. Complex features of modern languages like mutable variables, class, inheritance, exceptions, etc. must be considered by the underlying resource analysis used as a black box by our approach (and there are a number of approaches to handle them [2,5,11]). Thus they are handled implicitly in our setting. For the sake of simplicity, the set *Types* is defined as $\{\mathsf{int}\}$.

$$(1)\ \frac{r = eval(\mathsf{e}, tv), tr' = tr[y \mapsto \langle r, a_{pp}\rangle], H' = H \cup \{\!\!\{\langle id, y, a_{pp}, r\rangle\}\!\!\}}{\langle id, m, pp \equiv \mathsf{y} = \mathsf{acquire}\ (\mathsf{e}); s, tv, tr\rangle \cdot A; H \ \rightsquigarrow\ \langle id, m, s, tv, tr'\rangle \cdot A; H'}$$

$$(2)\ \frac{\langle r, a_{pp'}\rangle = tr(y), tr' = tr[y \mapsto \bot], H' = H \setminus \{\!\!\{\langle id, y, a_{pp'}, r\rangle\}\!\!\}}{\langle id, m, pp \equiv \mathsf{release}\ \mathsf{y}; s, tv, tr\rangle \cdot A; H \ \rightsquigarrow\ \langle id, m, s, tv, tr'\rangle \cdot A; H'}$$

Fig. 1. Language Semantics for resource allocation and release

We have *resource* variables used to refer to the resources allocated by an acquire instruction. A program consists of a set of methods whose definition takes the form $t\ m\ (t_1 v_1, \ldots t_n v_n)\{s\}$ where $t \in \mathit{Types}$ is the type returned by the method, $v_1, \ldots, v_n$ are the input parameters of types $t_1, \ldots, t_n \in \mathit{Types}$ and s is a sequence of instructions that adheres to the following grammar:

$$e ::= x \,|\, n \,|\, e + e \,|\, e * e \,|\, e - e \qquad b ::= e > e \,|\, e == e \,|\, b \wedge b \,|\, b \vee b \,|\, !b \qquad s ::= i \,|\, i; s$$
$$i ::= x{=}e \,|\, x{=}m(\overline{z}) \,|\, \mathsf{return}\ x \,|\, \mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \,|\, \mathsf{while}\ b\ \{s\} \,|\, y = \mathsf{acquire}\ (e) \,|\, \mathsf{release}\ y$$

We assume that resource variables, named y, are local to methods and they cannot be passed as input parameters nor returned by methods (otherwise tracking such references is more complex, while it is not relevant to the main ideas in the paper). We assume that the program includes a main($\overline{\mathsf{x}}$) method, where $\overline{\mathsf{x}}$ are the input parameters, from which the execution starts. The instruction y = acquire (e) allocates the amount of resources stated by the expression e. The instruction release y releases the resources allocated at the last acquire associated to y. If a resource variable is reused without releasing its resources, the reference to such resources is lost and they cannot be released any longer.

Example 1. Fig. 2 shows to the left a method m (abbreviation of main) that allocates resources at lines 2 (L2 for short) and L4. The resources allocated at L2 are released at L5. In addition, method m invokes method q at L3 and L6. For simplicity, we assume that m is called using positive values for n and s and the expressions k_1, k_2, k_3 are constant integer values. As it is not relevant, we do not include the return instruction at the end of the methods. Method q executes a while loop where k_2 units are allocated at L10 and such resources are not released. Thus, these resources *escape* from the scope of the loop and the method, i.e., they *leak* upon exit of the loop and return of the method. Besides, the program allocates w units at L11. As we have two calls to q, the input parameter w will take the value s or s+4. The resources allocated at L11 are released at L12 and do not escape from the loop execution. In addition, at L15 we have an additional, non-released, acquire of k_3 units.

A *program state* is of the form AS;H, where AS is a stack of *activation records* and H is a *resource handler*. Each activation record is of the form $\langle id, m, s, tv, tr\rangle$, where id is a unique identifier, m is the name of the method, s is the sequence of instructions to be executed, tv is a variable mapping and tr is a resource variable mapping. When resources are allocated in m, tr maps the corresponding resource variable to a tuple of the form $\langle r, a_{pp}\rangle$, where r is the amount of resources allocated and a_{pp} is the program point of the instruction where the resources have been allocated. The resource handler H is a multiset which stores

```
1  m (int n, int s){
2  ① x=acquire(k1);
3     q(n,s);
4  ② y=acquire(s);
5     release x;
6     q(n+2, s+4);
7  }
8  q (int i, int w){
9     while(i > 0) {
10 ③    z=acquire(k2);
11 ④    r=acquire(w);
12       release r;
13       i = i − 1;
14    }
15 ⑤ t=acquire(k3);
16 }
```

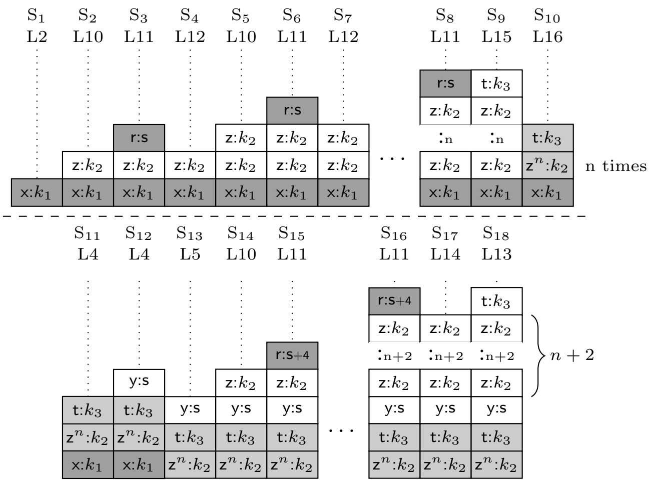

Fig. 2. Running Example

the resources allocated so far, containing elements of the form $\langle id, y, a_{pp}, r\rangle$, where id is the activation record identifier, y is the variable name, a_{pp} is the program point of the acquire and r is the amount of resources allocated. Fig. 1 shows, in a rewriting-based style, the rules that are relevant for the resource consumption. The semantics of the remaining instructions is standard. Intuitively, rule (1) evaluates the expression e and adds a new element to H. As H stores the resources allocated so far, it might contain identical tuples. Moreover, the resource variable mapping tr is updated with variable y linked to $\langle r, a_{pp}\rangle$. Rule (2) takes the information stored in tr for y, i.e. $\langle r, a_{pp}\rangle$, and removes from H one instance of the corresponding element. In addition, variable y is updated to point to $\perp$, which means that y does not have any resources associated. When the execution performs a release on a variable that maps to $\perp$ (because no acquire has been performed or because it has already been released), the resources state is not modified. Execution starts from a main method and an initial state $S_0 = \langle 0, \mathsf{main}, body(\mathsf{main}), tv(\overline{x}), \emptyset\rangle; \emptyset$, where $tv(\overline{x})$ is the variable mapping initialized with the values of the input parameters. Complete executions are of the form $S_0 \rightsquigarrow S_1 \rightsquigarrow \ldots \rightsquigarrow S_n$ where S_n corresponds to the last state. Infinite traces correspond to non-terminating executions.

Example 2. To the right of Fig. 2 we depict the evolution of the resources accumulated in H. We use S_i, to refer to the execution state i and, below each state, we include the program line which is executed at such state. For each state we show the elements stored in H but, for simplicity, we do not include in the figure the id nor a_{pp}. At S_1, H accumulates k_1 units due to the acquire at L2. S_2, S_3 and S_4 depict H along the first iteration of the loop, where k_2 units are acquired and not released from z. Moreover, within the loop, s units are acquired at L11 and released from r at L12. At S_5, which corresponds to the second iteration of the loop, we reuse the resource variable z and we have two identical elements in H. As the loop iterates n times, at the last iteration (S_9)

we have $(n{-}1){*}k_1$ units that have lost their reference. Additionally, k_3 extra units pointed by t are allocated at S_9. At S_{10}, which corresponds to the end of the execution of the method, $n{*}k_2{+}k_3$ units escape from the first execution of q and they are no longer available to be released. We represent such escaped resources with light grey color. For brevity, we use $\mathsf{z}^n{:}k_2$ to represent n instances of the element $\mathsf{z}{:}k_2$. At S_{12} we acquire s resources and we release the k_1 units pointed by x at S_{13}. At S_{14} we start a new execution of method q.

2.2 Definition of Peak Cost

Let us formally define the notion of *peak cost* in the concrete setting. The peak cost corresponds to the maximum amount of resources that are used simultaneously. We use H_i to refer to the multiset H at S_i, and we use R_i to denote the amount of resources contained in H_i, i.e., $R_i = \sum\{r \mid \langle\_,\_,\_,r\rangle \in H_i\}$. By '_', we mean any possible value. In the next definition, we use R_i to define the notion of *peak cost* for an execution trace.

Definition 1 (Concrete Peak Cost). *The* peak cost *of an execution trace* $t{\equiv}S_0 \leadsto S_n$ *of a program P on input values* $\overline{x}$ *is defined as* $\mathcal{P}(\overline{x}){=}max(\{R_i \mid S_i{\in}t\})$.

Example 3. According to the evolution of H shown to the right of Fig. 2, the maximum value of R_i could be reached at four different states, S_8, S_{12}, S_{16} and S_{18}. We ignore those states where H is subsumed by other states as they cannot be maximal. For instance, states S_1 to S_7 or S_9 are subsumed by S_8; or S_{12} contains S_{10}, S_{11} and S_{13}. Thus, $\mathcal{P}(n,s){=}max(R_8, R_{12}, R_{16}, R_{18})$, where $R_8 = k_1{+}n{*}k_2{+}s$, $R_{12} = k_1{+}n{*}k_2{+}k_3{+}s$, $R_{16} = n{*}k_2{+}k_3{+}s{+}(n{+}2){*}k_2{+}(s{+}4)$, and $R_{18}{=}n{*}k_2{+}k_3{+}s{+}(n{+}2){*}k_2{+}k_3$. Thus, the peak cost of the example depends not only on the input parameters n, s, but also on the values of k_1, k_2, k_3.

3 Simultaneous Resource Analysis

The *simultaneous resource analysis* (SRA) is used to infer the sets of acquire instructions that can be simultaneously in use. The abstract state of the SRA consists of two sets $\mathcal{C}$ and $\mathcal{H}$. The set $\mathcal{C}$ contains elements of the form $y{:}a_{pp}$ indicating that the resource variable y is linked to the acquire instruction at program point pp. Since it is not always possible to relate the acquire instruction to its corresponding resource variable, we use $\star{:}a_{pp}$ to represent that some resources have been acquired at a_{pp} but the analysis has lost the variable linked to a_{pp}. The set $\mathcal{H}$ is a set of sets, such that each set contains those a_{pp} that are simultaneously alive in an abstract state of the analysis. Let us introduce some notation. We use $\ddot{m}$ to refer to the program point after the return instruction of method m. We use $\mathcal{C}_{pp}$ (resp. $\mathcal{H}_{pp}$) to denote the value of $\mathcal{C}$ (resp. $\mathcal{H}$) after processing the instruction at program point pp. $\mathcal{A}(\mathcal{C})$ is the set $\{a_{pp} \mid \_{:}a_{pp} \in \mathcal{C}\}$ that contains all a_{pp} in $\mathcal{C}$. The operation $\mathcal{H}_1 \uplus \mathcal{H}_2$, where $\mathcal{H}_1$ and $\mathcal{H}_2$ are sets of sets, first applies $\mathcal{H} = \mathcal{H}_1 \cup \mathcal{H}_2$, and then removes those sets in $\mathcal{H}$ that are contained in another set in $\mathcal{H}$.

The analysis of each method m abstractly executes its instructions, by applying the transfer function τ in Fig. 3, such that the abstract state at each

$$\begin{array}{lr@{\;}l}
(1) & \tau(pp : \mathsf{y{=}acquire(\_)}, \langle\mathcal{C},\mathcal{H}\rangle) = & \langle \mathcal{C}[y{:}a_{pp'}/\star{:}a_{pp'}] \cup \{y{:}a_{pp}\}, \mathcal{H} \uplus \{\mathcal{A}(\mathcal{C}) \cup \{a_{pp}\}\}\rangle \\
(2) & \tau(pp : \mathsf{release\ y}, \langle\mathcal{C},\mathcal{H}\rangle) = & \langle \mathcal{C} \setminus \{y{:}a_{pp}\}, \mathcal{H}\rangle \\
(3) & \tau(pp : \mathsf{m(\_)}, \langle\mathcal{C},\mathcal{H}\rangle) = & \langle \mathcal{C} \cup \mathcal{C}_{\ddot{m}}[x{:}a_{pp'}/\star{:}a_{pp'}], \mathcal{H} \uplus \{\mathcal{A}(\mathcal{C}) \cup M \mid M \in \mathcal{H}_{\ddot{m}}\}\rangle \\
(4) & \tau(pp : b, \langle\mathcal{C},\mathcal{H}\rangle) = & \langle\mathcal{C},\mathcal{H}\rangle
\end{array}$$

Fig. 3. Transfer Function of the Simultaneous Resource Analysis

program point describes the status of all acquire instructions executed so far. The set $\mathcal{C}$ is used to infer the local effect of the acquire and release instructions within a method. The set $\mathcal{H}$ is used to accumulate the information of the acquire instructions that might have been in use simultaneously. Let us explain the different cases of the transfer function τ. The execution of acquire, case (1), links the acquire to the resource variable y by adding $\{y{:}a_{pp}\}$ to $\mathcal{C}$. As a resource variable can only point to one acquire instruction, in (1) we update any existing $y{:}a_{pp'}$ by removing the previous link to y and replacing it by $\star$. In addition, rule (1) performs the operation $\{\mathcal{A}(\mathcal{C}) \cup \{a_{pp}\}\} \uplus \mathcal{H}$ to capture in $\mathcal{H}$ the acquired resources simultaneously in use at this point. In (2) we remove the last acquire instruction pointed to by the resource variable y. When a method is invoked (rule (3)), we add to $\mathcal{C}$ those resources that might escape from m ($\mathcal{C}_{\ddot{m}}$) but replacing their resource variables in m by $\star$ (as resource variables are local). Additionally, at (3), all sets in $\mathcal{H}_{\ddot{m}}$ are joined with $\mathcal{A}(\mathcal{C})$ to capture the resources that might have been simultaneously alive in the execution of m. The resulting sets of such operation are added to $\mathcal{H}$. We define the $\sqcup$ operation between two abstract states $\langle\mathcal{C}_1,\mathcal{H}_1\rangle \sqcup \langle\mathcal{C}_2,\mathcal{H}_2\rangle$ as $\langle\mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{H}_1 \uplus \mathcal{H}_2\rangle$. The analysis of while loops requires iterating until a fixpoint is reached. As the number of acquire instructions and the number of resource variables in the program are finite, widening is not needed.

Example 4. Let us apply the SRA to the running example. To avoid cluttering the expressions, instead of the line numbers, we use a_i to refer to the acquire at the program point marked with ⓘ in Fig. 2. For instance, a_1 refers to the acquire marked with ① at L2. We use $\mathcal{C}_l$ (resp. $\mathcal{H}_l$) to denote the set $\mathcal{C}$ (resp. $\mathcal{H}$) at line l. Let us see the results of the SRA for some selected program points.

$$\begin{array}{ll}
\mathcal{C}_2 = \{x{:}a_1\} & \mathcal{H}_2 = \{\{a_1\}\} \\
\mathcal{C}_3 = \{x{:}a_1, \star{:}a_3, \star{:}a_5\} & \mathcal{H}_3 = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5\}\} \\
\mathcal{C}_4 = \{x{:}a_1, \star{:}a_3, \star{:}a_5, y{:}a_2\} & \mathcal{H}_4 = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}\} \\
\mathcal{C}_5 = \{\star{:}a_3, \star{:}a_5, y{:}a_2\} & \mathcal{H}_5 = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}\} \\
\mathcal{C}_6 = \mathcal{C}_{\ddot{m}} = \{\star{:}a_3, \star{:}a_5, y{:}a_2\} & \mathcal{H}_6 {=} \mathcal{H}_{\ddot{m}} {=} \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}, \{a_2, a_3, a_4, a_5\}^{\circledast}\} \\
\mathcal{C}_{10} = \{\star{:}a_3, z{:}a_3\} & \mathcal{H}_{10} = \{\{a_3, a_4\}\} \\
\mathcal{C}_{11} = \{\star{:}a_3, z{:}a_3, r{:}a_4\} & \mathcal{H}_{11} = \{\{a_3, a_4\}\} \\
\mathcal{C}_{12} = \mathcal{C}_{14} = \{\star{:}a_3, z{:}a_3\} & \mathcal{H}_{12} = \mathcal{H}_{14} = \{\{a_3, a_4\}\} \\
\mathcal{C}_{15} = \mathcal{C}_{\ddot{q}} = \{\star{:}a_3, z{:}a_3, t{:}a_5\} & \mathcal{H}_{15} = \mathcal{H}_{\ddot{q}} = \{\{a_3, a_4\}, \{a_3, a_5\}\}
\end{array}$$

We can see that $\mathcal{C}_{11}$ is the only program point where a_4 is alive as it is released at L12. On the contrary, as a_3 is not released within the loop, we include $\star{:}a_3$ in $\mathcal{C}_{10}{-}\mathcal{C}_{14}$, and it escapes from the loop and from q. As $\mathcal{H}$ gathers all a_{pp} that might be alive at any program point, when the fixpoint is reached, $\mathcal{H}_{10} - \mathcal{H}_{14}$ contain the set $\{a_3, a_4\}$. The computation of $\mathcal{H}_{\ddot{q}}$ is done by means of the operation $\mathcal{A}(\mathcal{C}_{\ddot{q}}) \uplus \mathcal{H}_{14}$, that is, $\mathcal{H}_{\ddot{q}}{=}\{\{a_3, a_5\}\} \uplus \{\{a_3, a_4\}\}{=}\{\{a_3, a_5\}, \{a_3, a_4\}\}$, capturing that a_3, a_4, a_5 are not simultaneously in use at any state of q. Moreover, we can see in $\mathcal{C}_{\ddot{\mathsf{q}}}$ that the resources allocated at a_3 and a_5 escape from the execution of q. Let us continue with the computation of $\mathcal{C}_3$ and $\mathcal{H}_3$. Firstly, $\star{:}a_3$ and $\star{:}a_5$ are added to $\mathcal{C}_3$. Secondly, $\mathcal{H}_3$ is computed by adding $\mathcal{C}_2{=}\{a_1\}$ to all sets in $\mathcal{H}_{\ddot{\mathsf{q}}}$. To compute $\mathcal{C}_4$, the analysis adds $y{:}a_2$ to $\mathcal{C}_3$. The computation of $\mathcal{H}_4$ adds $\{a_1, a_3, a_5, a_2\}$ to $\mathcal{H}_3$, and replaces $\{a_1, a_3, a_5\}$ because it is a subset of $\{a_1, a_3, a_5, a_2\}$. Finally, to obtain $\mathcal{H}_6$, the set $\mathcal{A}(\mathcal{C}_6){=}\{a_3, a_5, a_2\}$ is added to the sets in $\mathcal{H}_{\ddot{\mathsf{q}}}$, resulting in the set $T = \{\{a_2, a_3, a_4, a_5\}, \{a_2, a_3, a_5\}\}$. Then $\mathcal{H}_6$ is obtained by computing $\mathcal{H}_5 \uplus T$. Note that $\{a_2, a_3, a_5\}$ is not in $\mathcal{H}_6$ as it is contained in a set of $\mathcal{H}_5$.

Theorem 1 (Soundness). *Given an execution trace $t \equiv S_0 \rightsquigarrow \ldots \rightsquigarrow S_n$ of a program P on input values $\overline{x}$, for any state $S_i \in t$, we have that:*
(a) $\exists\ \mathbb{H} \in \mathcal{H}_{\ddot{\mathsf{main}}}.\ A(H_i) \subseteq \mathbb{H}$ where $A(H_i) = \{a_{pp} \mid \langle \_, \_, a_{pp}, \_ \rangle \in H_i\}$;
(b) if $\exists \langle \_, \_, a_{pp}, \_ \rangle \in H_n$ then $\_{:}a_{pp} \in \mathcal{C}_{\ddot{\mathsf{main}}}$

4 Non-cumulative Resource Analysis

In this section we present our approach to use the information obtained in Sec. 3 to infer the peak cost of the execution. The first part, Sec. 4.1, consists in performing a program-point resource analysis in which we are able to infer the resources acquired at the points of interest. In Sec. 4.2, we discard from the upper bound obtained before those resources which are not used simultaneously.

4.1 Program-Point Resource Analysis

Our goal is to distinguish within the upper bounds (UB) obtained by resource analysis the amount of resources acquired at a given program point. To do so, we rely on the notion of *cost center* (CC) [1]. Originally, CCs were introduced for the analysis of distributed systems, such that, each CC is a symbolic expression of the form $c(o)$ where o is a location identifier used to separate the cost of each distributed location. Essentially, the resource analysis assigns the cost of an instruction *inst* to the distributed location o by multiplying the cost due to the execution of the instruction, denoted $cost(inst)$ in a generic way, by the cost center of the location $c(o)$, i.e., $cost(inst){*}c(o)$. This way, the UBs that the analysis obtains are of the form $\sum c(o_i){*}C_i$, where each o_i is a location identifier and C_i is the total cost accumulated at this location.

Importantly, the notion of CC can be used in a more general way to define the granularity of a cost analyzer, i.e., the kind of separation that we want to observe in the UBs. In our concrete application, the expressions of the cost centers o_i will refer to the program points of interest. Thus, we are defining a resource analyzer that provides the resource consumption at program point level, i.e., a *program point* resource analysis. In particular, we define a CC for each $\mathsf{acquire}$ instruction in the program. Thus, CCs are of the form $c(a_{pp})$ for each instruction $pp{:}\mathsf{acquire(e)}$. In essence, the analyzer every time that accounts for the cost of executing an $\mathsf{acquire}$ instruction multiplies such cost by its corresponding cost center. The amount of resources allocated at the instruction $pp{:}\mathsf{acquire(e)}$ is accumulated as an expression of the form $c(a_{pp})*\mathsf{nat}(e)$, where $\mathsf{nat}(e)$ is a function that returns e if $e>0$ and 0 otherwise. We wrap the expression e with nat because this way the analyzer treats it as a non-negative expression whose cost we want to maximize, and computes the worst case of such expression (technical details can be found in [2]). The cost analyzer computes an *upper bound* for the total cost of executing P as an expression of the form $\mathcal{U}_P(\bar{x})=\sum_{i=1}^{n} c(a_i)*C_i$, where C_i is a cost expression that bounds the resources allocated by the acquire instructions of the program. We omit the subscript in $\mathcal{U}$ when it is clear from the context. If one is interested in the amount of resources allocated by one particular acquire instruction a_{pp}, denoted $\mathcal{U}(\bar{x})|_{a_{pp}}$, we simply replace all $c(a_{pp'})$ with $pp \neq pp'$ by 0 and $c(a_{pp})$ by 1. We extend it to sets as $\mathcal{U}(\bar{x})|_S = \sum_{a_{pp}\in S} \mathcal{U}(\bar{x})|_{a_{pp}}$.

Example 5. The program point UB for the running example is:

$$\mathcal{U}(n,s) = \overbrace{c(a_1)*k_1}^{e_1} + \overbrace{c(a_2)*\mathsf{nat}(s)}^{e_2} + \overbrace{\mathsf{nat}(n) * (c(a_3)*k_2 + c(a_4)*\mathsf{nat}(s)) + c(a_5)*k_3}^{e_3} + \underbrace{\mathsf{nat}(n{+}2) * (c(a_3)*k_2 + c(a_4)*\mathsf{nat}(s{+}4)) + c(a_5)*k_3}_{e_4}$$

We have a CC for each acquire instruction in the program multiplied by the amount of resources allocated by the corresponding acquire. In the examples, we do not wrap constants in nat because constant values do not need to be maximized, e.g. in the subexpression e_1 which corresponds to the cost of L2. The subexpression e_2 corresponds to L4 where s units are allocated. Expression e_3 corresponds to the first call to q, where the loop iterates $\mathsf{nat}(n)$ times and consumes $c(a_3)*k_2$ ($L10$) and $c(a_4)*\mathsf{nat}(s)$ ($L11$) resources for each iteration, plus the final acquire at $L15$, which allocates $c(a_5)*k_3$ resources. The cost of the second call to q is captured by e_4, where the number of iterations is bounded by $\mathsf{nat}(n{+}2)$ and $\mathsf{nat}(s{+}4)$ resources are allocated. e_4 also includes the cost allocated at L15. Let us continue by using $\mathcal{U}(n,s)$ to compute the resources allocated at a particular location, e.g. a_4, denoted by $\mathcal{U}(n,s)|_{a_4}$. To do so, we replace $c(a_4)$ by 1 and the rest of $c(\_)$ by 0. Thus, $\mathcal{U}(n,s)|_{a_4} = \mathsf{nat}(n)*\mathsf{nat}(s)+\mathsf{nat}(n{+}2)*\mathsf{nat}(s{+}4)$. Similarly, given the set of program points $\{a_3, a_5\}$, we have $\mathcal{U}(n,s)|_{\{a_3,a_5\}} = \mathcal{U}(n,s)|_{\{a_3\}} + \mathcal{U}(n,s)|_{\{a_5\}} = \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3$.

4.2 Inference of Peak Cost

We can now put all pieces together. The SRA described in Sec. 3 allows us to infer the acquire instructions which could be allocated simultaneously. Such information is gathered in the set $\mathcal{H}$ of the SRA. In fact, the set $\mathcal{H}$ at the last program point of the program, namely $\ddot{\mathsf{main}}$, collects all possible states of the resource allocation during program execution. Using this set we define the notion of *peak cost* as the maximum of the UBs computed for each possible set in $\mathcal{H}_{\ddot{\mathsf{main}}}$.

Definition 2 (Peak Cost). *The* peak cost *of a program $P(\overline{x})$, denoted $\widehat{\mathcal{P}}(\overline{x})$, is defined as $\widehat{\mathcal{P}}(\overline{x}) = max(\{\mathcal{U}(\overline{x})|_{\mathbb{H}} \mid \mathbb{H} \in \mathcal{H}_{\ddot{\mathsf{main}}}\})$.*

Intuitively, for each $\mathbb{H}$ in $\mathcal{H}_{\ddot{\mathsf{main}}}$, we compute its restricted UB, $\mathcal{U}(\overline{x})|_{\mathbb{H}}$, by removing from $\mathcal{U}(\overline{x})$ the cost due to acquire instructions that are not in $\mathbb{H}$, i.e., those acquire that were not active simultaneously with the elements in h.

Example 6. By using $\mathcal{H}_{\ddot{\mathsf{m}}} = \{\{a_1, a_3, a_4\}, \{a_1, a_3, a_5, a_2\}, \{a_2, a_3, a_4, a_5\}\}$, the peak cost of m is the maximum of the expressions:

$$\begin{aligned}
\mathcal{U}(n,s)|_{\{a_1,a_3,a_4\}} &= k_1 + \mathsf{nat}(n)*(k_2 + \mathsf{nat}(s)) + \mathsf{nat}(n{+}2)*(k_2 + \mathsf{nat}(s{+}4)) \\
\mathcal{U}(n,s)|_{\{a_1,a_3,a_5,a_2\}} &= k_1 + \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3 \\
\mathcal{U}(n,s)|_{\{a_2,a_3,a_4,a_5\}} &= \mathsf{nat}(s){+}\mathsf{nat}(n)*(k_2{+}\mathsf{nat}(s)) + k_3 + \mathsf{nat}(n{+}2)*(k_2{+}\mathsf{nat}(s{+}4)){+}k_3
\end{aligned}$$

Each UB expression over-approximates the value of R for the different states seen in Ex. 3 that could determine the concrete peak cost, namely $\mathcal{U}(n,s)|_{\{a_1,a_3,a_4\}}$ over-approximates the resource consumption at state S_8, $\mathcal{U}(n,s)|_{\{a_1,a_3,a_5,a_2\}}$ corresponds to S_{12}, and $\mathcal{U}(n,s)|_{\{a_2,a_3,a_4,a_5\}}$ bounds S_{16} and S_{18}.

Theorem 2 (Soundness). $\mathcal{P}(\overline{x}) \leq \widehat{\mathcal{P}}(\overline{x})$.

5 Extensions of the Basic Framework

In this section we discuss several extensions to our basic framework. First, Sec. 5.1 discusses how context-sensitive analysis can improve the accuracy of the results. Sec. 5.2 describes an improvement for handling *transient* acquire instructions, i.e., those resources which are allocated and released repeatedly but only one of all allocations is in use at a time. Finally, Sec. 5.3 introduces the extension of the framework to handle several kinds of resources.

5.1 Context-Sensitivity

Establishing the granularity of the analysis at the level of program points may lead to a loss of precision. This is because the computation of the SRA and the resource analysis are not able to distinguish if an acquire instruction is executed multiple times from different contexts. As a consequence, all resource usage associated to a given a_{pp} is accumulated in a single CC.

Example 7. The set $\mathcal{H}_{\ddot{m}}$ computed in Ex. 4 includes a_4 in two different sets. The first set corresponds to the first call to q (L3), where s units are allocated, whereas the second set corresponds to the second call (L6), and where s+4 units are allocated. Observe that the SRA of m does not distinguish such situation as both executions of L11 are represented as a single program point a_4. The same occurs in the computation of the UBs. In Ex. 6 we have computed $\mathcal{U}(n,s)|a_4 = \mathsf{nat}(n)*\mathsf{nat}(s){+}\mathsf{nat}(n{+}2)*\mathsf{nat}(s{+}4)$, which accounts for the resources acquired at L11. Note that $\mathcal{U}(n,s)|a_4$ does not separate the cost of the different calls to q.

Intuitively, this loss of precision can be detected by checking if the *call graph* of the program contains convergence nodes, i.e., methods that have more than one incoming edge because they are invoked from different contexts. In such case, we can use standard techniques for context-sensitive analysis [16], e.g., method replication. In particular, the program can be rewritten by creating a different copy of the method for each incoming edge. Method replication guarantees that the calling contexts are not merged unless they correspond to a method call within a loop (or transitively from a loop). In the latter case, we indeed need to merge them and obtain the worst-case cost of all iterations, as the underlying resource analysis [2] already does.

Example 8. As q is called at L3 and L6, the application of the context-sensitive replication builds up a program with two methods: $\mathsf{q\_1}$ (from the call at L3) and $\mathsf{q\_2}$ (from L6). In addition, the modified version of m, denoted m', calls $\mathsf{q\_1}$ at L3 and $\mathsf{q\_2}$ at L6. We use a_{31} (resp. a_{32}) to refer to the $\mathsf{acquire}$ at L10 for the replica $\mathsf{q\_1}$ (resp. $\mathsf{q\_2}$). The SRA for m' returns: $\mathcal{H}_{\ddot{\mathsf{m}}'} = \{\{a_1, a_{31}, a_{41}\}, \{a_1, a_{31}, a_{51}, a_2\}, \{a_{31}, a_{51}, a_2, a_{32}, a_{42}\}, \{a_{31}, a_{51}, a_2, a_{32}, a_{52}\}\}$ and $\mathcal{C}_{\ddot{\mathsf{m}}'} = \{a_2, a_{31}, a_{32}, a_{51}, a_{52}\}$. Observe that the set marked with $\circledast$ in Ex. 4 is now split in two different sets, which precisely capture the states S_{16} and S_{18} of Fig. 2. Moreover, we distinguish a_{41}, a_{42} and a_{51}, a_{52} that allow us to separate the different calls to q, which is crucial for accounting the peak cost more accurately. The UB for m' is:

$$\mathcal{U}_{\mathsf{m}'}(n,s) = c(a_1)*k_1 + c(a_2)*\mathsf{nat}(s) + \mathsf{nat}(n)*(c(a_{31})*k_2 + c(a_{41})*\mathsf{nat}(s)) + c(a_{51})*k_3 + \mathsf{nat}(n{+}2)*(c(a_{32})*k_2 + c(a_{42})*\mathsf{nat}(s{+}4)) + c(a_{52})*k_3$$

In contrast to $\mathcal{U}_m(n,s)|_{a_4}$, shown in Ex. 5, now we can compute $\mathcal{U}_{m'}(n,s)|_{a_{41}} = \mathsf{nat}(n)*\mathsf{nat}(s)$ and $\mathcal{U}_{m'}(n,s)|_{a_{42}} = \mathsf{nat}(n{+}2)*\mathsf{nat}(s{+}4)$. $\widehat{\mathcal{P}}_{m'}(n,s)$ is the maximum of:

$$\begin{array}{rll}
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{41}\}} &= k_1 + \mathsf{nat}(n)*(k_2 + \mathsf{nat}(s)) & [S_8] \\
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{51},a_2\}} &= k_1 + \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 & [S_{12}] \\
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{42}\}} &= \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*(k_2{+}\mathsf{nat}(s{+}4)) & [S_{16}] \\
\mathcal{U}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{52}\}} &= \mathsf{nat}(s) + \mathsf{nat}(n)*k_2 + k_3 + \mathsf{nat}(n{+}2)*k_2 + k_3 & [S_{18}]
\end{array}$$

To the right of the UB expressions above we show their corresponding state of Fig. 2. In contrast to Ex. 6, now we have a one-to-one correspondence, and thus $\widehat{\mathcal{P}}_{m'}(n,s)$ is more accurate than $\widehat{\mathcal{P}}_m(n,s)$ in Ex. 6.

5.2 Handling Transient Resource Allocations

A complementary optimization with that in Sec. 5.1 can be performed when resources are acquired and released multiple times along the execution of the program within loops (or recursion). We use the notion of *transient* $\mathsf{acquire}$ to refer to an $\mathsf{acquire}(e)$ instruction at a_{pp} that is executed and released repeatedly but in such a way that the resources allocated by different executions of a_{pp} never coexist. As the UBs of Sec. 4 are computed by multiplying the number of times that each $\mathsf{acquire}$ instruction is executed by the worst case cost of each execution, the fact that the allocations of a transient $\mathsf{acquire}$ do not coexist is not accurately captured by the UB.

Example 9. Let us focus on the $\mathsf{acquire}$ a_4 of the running example. Although a_4 is executed multiple times within the loop, each allocation does not escape from the corresponding iteration because it is released at L12. To the right of Fig. 2 we can see that states S_3, S_6, S_8, S_{15} and S_{16} include the cost allocated by a_4 only once (elements in dark grey). Thus, a_4 is a transient $\mathsf{acquire}$. In spite of this, we compute $\mathcal{U}_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(n)*\mathsf{nat}(s)$, which accounts for the cost allocated at a_{41} as many times as a_{41} might be executed. Certainly, $\mathcal{U}_{\mathsf{m}'}(n,s)|_{a_{41}}$ is a sound but imprecise approximation for the cost allocated by a_{41}.

We can improve the accuracy of the UBs for a transient $\mathsf{acquire}$ a_{pp} by including its worst case cost only once. We start by identifying when a_{pp} is transient in the concrete setting. Intuitively, if a_{pp} is transient the resources allocated at a_{pp} do not leak. Thus, in the last state of the execution, S_n, no resource allocated at a_{pp} remains in H_n (see the semantics at Fig. 1).

Definition 3 (Transient Acquire). *Given a program P, an* acquire *instruction a_{pp} is* transient *if for every execution trace of P, $S_1 \rightsquigarrow \ldots \rightsquigarrow S_n$, $\langle \_, \_, a_{pp}, \_\rangle \notin H_n$.*

Example 10. In Fig. 2 we can see that a_1 and a_4 (shown in dark grey) are transient because their resources are always released at L5 and L12, resp.

In order to count the cost of a transient acquire only once, we use a particular instantiation of the cost analysis described in Sec. 4.1 to determine an UB on the number of times that such acquire might be executed. We use $\mathcal{U}^c$ to denote such UB which is computed by replacing the expression C_i (see Sec. 4.1) by 1 in the computation of $\mathcal{U}$. Assuming that $\mathcal{U}$ and $\mathcal{U}^c$ have been approximated by the same cost analyzer, we gain precision by obtaining the cost associated to a transient acquire instruction using its *singleton cost.*

Definition 4 (Singleton Cost). *Given a_{pp} we define its* singleton cost *as $\widetilde{\mathcal{U}}(\overline{x})|_{a_{pp}} = \mathcal{U}(\overline{x})|_{a_{pp}} / \mathcal{U}^c(\overline{x})|_{a_{pp}}$ if $\_{:}a_{pp} \notin \mathcal{C}_{\ddot{\mathsf{main}}}$ and $\widetilde{\mathcal{U}}(\overline{x})|_{a_{pp}} = \mathcal{U}(\overline{x})|_{a_{pp}}$, otherwise.*

Intuitively, when a_{pp} is transient, its singleton cost is obtained dividing the accumulated UB by the number of times that a_{pp} is executed. If it is not transient, we must keep the accumulated UB. According to Def. 3 and Th. 1(b), if $a_{pp} \notin \mathcal{C}_{\ddot{\mathsf{main}}}$, then a_{pp} is transient, and so we can perform the division. We use $\widetilde{\mathcal{P}}$ to refer to the peak cost obtained by using $\widetilde{\mathcal{U}}$ instead of $\mathcal{U}$. In general, given a set of a_{pp}, we use $\widetilde{\mathcal{U}}_{\mathsf{m}'}|_S$ to refer to the UBs computed using the singleton cost of each $a_{pp} \in S$.

Example 11. Let us continue with the context-sensitive replica of the running example, m'. We start by computing $\mathcal{U}^c_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(n)$ and $\mathcal{U}_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(n) * \mathsf{nat}(s)$. As we can see in Ex. 8, $a_{41}, a_{42} \notin \mathcal{C}_{\ddot{m}'}$, then $\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{a_{41}} = \mathsf{nat}(s)$ which is the worst case of executing a_{41} only once. For a_{42} we have $\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{a_{42}} = \mathsf{nat}(s{+}4)$. Regarding the remaining acquire instructions, either they cannot be divided, or can be divided by 1. Thus, we have that $\widetilde{\mathcal{P}}_{\mathsf{m}'}(n,s)$ is the maximum of the following expressions:

$$\begin{aligned}
\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{41}\}} &= k_1 + \mathsf{nat}(n){*}k_2 + \mathsf{nat}(s) && [S_8]\\
\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_1,a_{31},a_{51},a_2\}} &= k_1 + \mathsf{nat}(s) + \mathsf{nat}(n){*}k_2 + k_3 && [S_{12}]\\
\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{42}\}} &= \mathsf{nat}(s) + \mathsf{nat}(n){*}k_2 + k_3 + \mathsf{nat}(s{+}4) && [S_{16}]\\
\widetilde{\mathcal{U}}_{\mathsf{m}'}(n,s)|_{\{a_{31},a_{51},a_2,a_{32},a_{52}\}} &= \mathsf{nat}(s) + \mathsf{nat}(n){*}k_2 + k_3 + \mathsf{nat}(n{+}2){*}k_2 + k_3 && [S_{18}]
\end{aligned}$$

Theorem 3 (Soundness). *Given a program $P(\overline{x})$ and its context-sensitive replica $P'(\overline{x})$, we have that $\mathcal{P}_P(\overline{x}) \le \widetilde{\mathcal{P}}_{P'}(\overline{x})$.*

5.3 Handling Different Resources Simultaneously

Our goal is now to allow allocation of different types of resources in the program (e.g., we want to infer the heap space usage and the number of simultaneous connections to a database). To this purpose, we extend the instruction acquire(e) (see Sec. 2.1) with an additional parameter which determines the kind of resource to be allocated, i.e., acquire(res,e). Such extension does not require any modification to the semantics. We define the function $type(a_{pp})$ which returns the type of resource allocated at a_{pp}. Now, we extend Def. 1 to consider the resource of interest. We use $R_i(\mathsf{res})$ to refer to the following value $R_i(\mathsf{res}) = \sum\{r \mid \langle \_, \_, a_{pp}, r\rangle \in H_i \wedge type(a_{pp}) = \mathsf{res}\}$.

Definition 5 (Concrete Peak Cost). *Given a resource* res*, the* peak cost *of an execution trace* t *of program* $P(\overline{x}, \mathsf{res})$ *is* $\mathcal{P}(\overline{x}, \mathsf{res}) = max(\{R_i(\mathsf{res}) | S_i \in t\})$.

Interestingly, such extension does not require any modification neither to the SRA of Sec. 3 nor to the program point resource analysis of Sec. 4. This is due to the fact that the analysis works at the level of program points and one program point can only allocate one particular type of resource. We define $\mathcal{R}(\mathsf{res})$ as the set of program points that allocate resources of type res, i.e., $\mathcal{R}(\mathsf{res})=\{a_{pp} \mid type(a_{pp})=\mathsf{res}\}$. Thus, we extend the notion of peak cost of Def. 2 with the type of resource, i.e., $\widehat{\mathcal{P}}(\overline{x}, \mathsf{res})=max(\{\mathcal{U}(\overline{x})|_{\mathbb{H}\cap\mathcal{R}(\mathsf{res})} \mid \mathbb{H} \in \mathcal{H}_{\ddot{\mathsf{main}}}\})$. Observe that the only difference with Def. 2 is in the intersection $\mathbb{H} \cap \mathcal{R}(\mathsf{res})$ which restricts the considered $\mathsf{acquire}$ when computing the UBs. One relevant aspect is that by computing the UB only once, we are able to obtain the peak cost for different types of resources by restricting the UB for each resource of interest. The extension of Th. 2 and Th. 3 to include a particular resource is straightforward.

Example 12. Let us modify the $\mathsf{acquire}$ instructions of the running example in Fig. 2 to add the resource to be allocated. Now we have that L2 is $\mathsf{x} = \mathsf{acquire}(\mathsf{hd},k_1)$ and L11 is $\mathsf{r} = \mathsf{acquire}(\mathsf{hd},\mathsf{w})$, where hd is a type of resource. We assume that L4, L10, L15 allocate a different type of resource, e.g. a resource of type mem. Then, using the context-sensitive replica of the program, we have that $\mathcal{R}(\mathsf{hd}) = \{a_1, a_{41}, a_{42}\}$, and $\mathcal{R}(\mathsf{mem}) = \{a_2, a_{31}, a_{32}, a_{51}, a_{52}\}$. Now, using the UB from Ex. 11, we have that $\widehat{\mathcal{P}}(n, s, \mathsf{hd})_{\mathsf{m}'}$ is the maximum of the expressions:

$$\begin{array}{rll}
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_1,a_{31},a_{41}\}\cap\mathcal{R}(\mathsf{hd})} &= k_1+\mathsf{nat}(s) & [S_8]\\
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_1,a_{31},a_{51},a_2\}\cap\mathcal{R}(\mathsf{hd})} &= k_1 & [S_{12}]\\
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_{31},a_{51},a_2,a_{32},a_{42}\}\cap\mathcal{R}(\mathsf{hd})} &= \mathsf{nat}(s+4) & [S_{16}]\\
\widetilde{\mathcal{U}}_{m'}(n, s, \mathsf{hd})|_{\{a_{31},a_{51},a_2,a_{32},a_{52}\}\cap\mathcal{R}(\mathsf{hd})} &= 0 & [S_{18}]
\end{array}$$

6 Experimental Evaluation

We have implemented a prototype peak cost analyzer for simple sequential programs that follow the syntax of Sec. 2.1, but that besides use a functional language to define data types (the use of functions does not require any conceptual modification to our basic analysis). This language corresponds to the sequential sublanguage of ABS [13], a language which besides has concurrency features that are ignored by our analyzer. To perform the experiments, our analyzer has been applied to some programs written in ABS: BBuffer, a bounded-buffer for communicating producers and consumers; MailServer, a client-server system; Chat, a chat application; DistHT, an implementation of a hash table; BookShop, a book shop application; and PeerToPeer, a peer-to-peer network.

The non-cumulative resource that we measure is the peak of the size of the stack of activation records. For each method executed, an activation record is created, and later removed when the method terminates. The size might depend on the arguments used in the call, as due to the use of functional data structures, when a method is invoked, the data structures (used as parameters) are passed and stored. This aspect is interesting because we can measure the peak size, not

Table 1. Experimental Evaluation (times in seconds)

Benchmark	$\#_l$	$\#_e$	$\mathbf{T}_n$	$\mathbf{T}_c$	$\%_n$	$\%_c$	$\%_s$	$\%_{cn}$	$\%_{sn}$	$\%_{sc}$
BBuffer	105	3125	0.93	1.07	4.9	35.7	43.9	32.1	40.6	15.7
MailServer	115	3375	9.58	1.23	16.0	42.4	58.2	30.2	47.1	27.6
Chat	302	2500	0.58	0.58	69.9	69.9	92.9	0.0	74.8	74.8
DistHT	353	2500	0.68	2.27	40.2	82.8	84.8	71.2	74.6	10.7
BookShop	353	4096	2.22	2.41	6.5	6.5	32.4	0.0	27.9	27.9
PeerToPeer	240	4.09	5.62	11.86	0.4	8.8	11.4	8.5	11.1	3.0
					23.0	41.0	53.9	23.7	46.0	26.6

only due to activation records whose size is constant, but also measure the size of the data structures used in the invocations, and take them into account.

In order to evaluate our analysis we have obtained different UBs on the size of the stack of activation records and compared their precision. In particular, we have compared the UBs obtained by the resource analysis of [2] (a cumulative cost analyzer), our basic non-cumulative approach (Sec. 4.2), the context-sensitive extension of Sec. 5.1 and the UBs obtained by using the singleton cost of each acquire as described in Sec. 5.2. In order to obtain concrete values for the gains, we have evaluated the UB expressions for different combinations of the input arguments and computed the average. For a concrete input arguments $\overline{x}$, we compute the gain of $\widehat{\mathcal{P}}(\overline{x})$ w.r.t. $\mathcal{U}(\overline{x})$ using the formula $(1-\widehat{\mathcal{P}}(\overline{x})/\mathcal{U}(\overline{x}))*100$. In order to compute the sizes of the activation records of the methods, we have modified each method of the benchmarks by including in the beginning of the method one acquire and one release at the end of each method to free it. Let us illustrate it with an example, if we have a method Int m (Data d,Int i) {Int j=i+1}, we modify it to {x=acquire(1+1+d+1+1); Int j=i+1; release x;}. The addends of the expression 1+1+d+1+1 correspond to: the pointer to the activation record, the size of the returned value (1 unit), the size of the information received through d (d units), the size of i (1 unit), and the size of j (1 unit). The instruction release(x) releases all resources. Experiments have been performed on an Intel Core i5 (1.8GHz, 4GB RAM), running OSX 10.8.

Table 1 summarizes the results obtained. Columns $\#_l$ and $\#_e$ show, resp., the number of lines of code and the number of input argument combinations evaluated. Columns $\mathbf{T}_n$, $\mathbf{T}_c$ show, resp., the time (in seconds) to perform the basic non-cumulative analysis and the context-sensitive non-cumulative analysis. Columns $\%_n$, $\%_c$, $\%_s$ show, resp., the gain of the non-cumulative resource analysis, its context-sensitive extension and the singleton cost extension w.r.t. the cumulative analysis. Column $\%_{cn}$ shows the gain of $\widehat{\mathcal{P}}$ applied to the context sensitive replica of the program w.r.t. its application to the original program. Columns $\%_{sn}$ and $\%_{sc}$ show, resp., the gain of $\widetilde{\mathcal{P}}$ w.r.t. $\widehat{\mathcal{P}}$, and w.r.t. $\widehat{\mathcal{P}}$ applied to the context sensitive replica of the program. The last row shows the average of the results. As regards analysis times, we argue that the time taken by the analyzer is reasonable and the context-sensitive approach although more expensive is feasible. As regards precision, we can observe that the gains obtained by the non-cumulative analyses are significant w.r.t. the cumulative resource analysis. As it can be expected, $\widetilde{\mathcal{P}}$ shows the best results with gains from 11% to 93%.

The non-cumulative analysis and its context-sensitive version also present significant gains, on average 23% and 41% respectively. The improvement gained by applying non-cumulative analysis to the context-sensitive extension is also relevant, a gain of 23.7%. As resources are released in all methods, we achieve a significant improvement with $\widetilde{\mathcal{P}}$, from 46% to 26.6% on average. All in all, we argue that the experimental evaluation shows the accuracy of non-cumulative resource analysis and the precision gained with its extensions.

7 Conclusions and Related Work

To the best of our knowledge, this is the first generic framework to infer the peak of the resource consumption of sequential imperative programs. The crux of the framework is an analysis to infer the resources that might be used simultaneously along the execution. This analysis is formalized as a data-flow analysis over a finite domain of sets of resources. The inference is followed by a program-point resource analysis which defines the resource consumption at the level of the program points at which resources are acquired.

Previous work on non-cumulative cost analysis of sequential imperative programs has been focused on the particular resource of memory consumption with garbage collection, while our approach is generic on the kind of non-cumulative cost that one wants to measure. Our framework can be used to redefine previous analyses of heap space usage [5] into the standard cost analysis setting. Depending on the particular garbage collection strategy, the release instruction will be placed at one point or another. For instance, if one uses scope-based garbage collection, all release instructions are placed just before the method return instruction and our framework can be applied. If one wants to use a liveness-based garbage collection, then the liveness analysis determines where the release instructions should go, and our analysis is then applied. The important point to note is that these analyses [5] provided a solution based on the generation of non-standard cost relations specific to the problem of memory consumption. It thus cannot be generalized to other kind of non-cumulative resources. Non-cumulative resource analysis, by means of the use of malloc and free, is studied at [9], but the approach is limited to constant resource consumption. Several analyses around the RAML tool [12] also assume the existence of acquire and release instructions and the application of our framework to this setting is an interesting topic for further research. The differences between amortized cost analysis and a standard cost analysis are discussed in [6,10]. Also, we want to study the recasting of [7] into our generic framework.

Recent work defines an analysis to infer the peak cost of distributed systems [3]. There are two fundamental differences with our work: (1) [3] is developed for cumulative resources, and the extension to non-cumulative resources is not studied there and (2) [3] considers a concurrent distributed language, while our focus is on sequential programs. There is nevertheless a similarity with our work in the elimination from the total cost of elements that do not happen simultaneously. However, in the case of [3] this information is gathered by a complex may-happen-in-parallel analysis [4] which infers the interleavings that may occur during the execution followed by a post-process in which a graph is built and its cliques are used to detect when several tasks can be executing concurrently. In our case, we are able to detect when resources are used simultaneously by means of a simpler analysis defined as a standard data-flow analysis on a finite domain. Besides, the upper bounds in [3] are obtained by a task-level resource analysis since in their case they want to obtain the resource consumption at the granularity of tasks rather than at program point granularity. As in our case, the use of context sensitive analysis [16] can improve the accuracy of the results.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish project TIN2012-38137, and by the CM project S2013/ICE-3006.

References

1. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: Static Analyzer for Concurrent Objects. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 562–567. Springer, Heidelberg (2014)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Correas, J., Román-Díez, G.: Peak Cost Analysis of Distributed Systems. In: Müller-Olm, M., Seidl, H. (eds.) Static Analysis. LNCS, vol. 8723, pp. 18–33. Springer, Heidelberg (2014)
4. Albert, E., Flores-Montoya, A.E., Genaim, S.: Analysis of May-Happen-in-Parallel in Concurrent Objects. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 35–51. Springer, Heidelberg (2012)
5. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Parametric Inference of Memory Requirements for Garbage Collected Languages. In: ISMM 2010, pp. 121–130 (2010)
6. Alonso-Blas, D.E., Genaim, S.: On the Limits of the Classical Approach to Cost Analysis. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 405–421. Springer, Heidelberg (2012)
7. Braberman, V., Fernández, F., Garbervetsky, D., Yovine, S.: Parametric Prediction of Heap Memory Requirements. In: ISMM 2008, pp. 141–150. ACM (2008)
8. Braberman, V.A., Garbervetsky, D., Hym, S., Yovine, S.: Summary-based inference of quantitative bounds of live heap objects. SCP 92, 56–84 (2014)
9. Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. In: FMCAD 2009, pp. 205–212 (2009)
10. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 275–295. Springer, Heidelberg (2014)
11. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In: POPL 2009, pp. 127–139. ACM (2009)
12. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL 2013, pp. 185–197. ACM (2003)

13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
14. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 745–761. Springer, Heidelberg (2014)
15. Trinder, P.W., Cole, M.I., Hammond, K., Loidl, H.W., Michaelson, G.: Resource analyses for parallel and distributed coordination. CCPE 25(3), 309–348 (2013)
16. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: PLDI, pp. 131–144. ACM (2004)