Resource Analysis: From Sequential to Concurrent and Distributed Programs

Elvira Albert[1], Puri Arenas[1], Jesús Correas[1], Samir Genaim[1], Miguel Gómez-Zamalloa[1], Enrique Martin-Martin[1], Germán Puebla[2], and Guillermo Román-Díez[2]

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

Abstract. Resource analysis aims at automatically inferring *upper/lower bounds* on the worst/best-case cost of executing programs. Ideally, a resource analyzer should be parametric on the *cost model*, i.e., the type of cost that the user wants infer (e.g., number of steps, amount of memory allocated, amount of data transmitted, etc.). The inferred upper bounds have important applications in the fields of program optimization, verification and certification. In this talk, we will review the basic techniques used in resource analysis of sequential programs and the new extensions needed to handle concurrent and distributed systems.

1 Introduction

One of the most important characteristics of a program is the amount of resources that its execution will require, i.e., its *resource consumption*. Resource analysis (a.k.a. cost analysis [23]) aims at *statically* bounding the cost of executing programs for any possible input data value. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. Resource usage information has many applications, both during program development and deployment. *Upper bounds* on the worst-case cost are useful because they provide *resource guarantees*, i.e., it is ensured that the execution of the program will never exceed the amount of resources inferred by the analysis. *Lower bounds* on the best-case cost have applications in program parallelization, they can be used to decide if it is worth executing locally a task or requesting remote execution. Therefore, automated ways of estimating resource usage are quite useful and the general area of resource analysis has received [23,14,22] and is nowadays receiving [6,15,16,17] considerable attention. In this paper, we describe the main components underlying resource analysis of a today's imperative programming language, e.g., such techniques have been applied to analyze the resource consumption of sequential Java and Java bytecode [19]. In a next step, we describe the extension of the sequential framework to handle concurrent programs and overview the new notions of cost that arise in these contexts.

The rest of the paper is organized in four sections as follows:

- *Sequential.* Section 2 considers a minimalistic imperative language and summarizes the process of, from a program, generating *upper bounds* on the worst-case cost of executing the program in terms of the input data sizes. We also discuss relevant extensions of the basic framework to handle object-oriented programs and non-cumulative resources.

- *Distribution and Concurrency.* Section 3 describes the extension of such techniques to analyze distributed and concurrent programs. First, in Section 3.1, we introduce the basic instructions for *distribution*, namely to create distributed locations and to spawn an asynchronous task in a remote location; and for concurrency, in particular an instruction to synchronize with the termination of an asynchronous task and be able to release the processor if the task has not terminated yet (in this case, another task waiting in this location can take the processor). In Sections 3.2 and 3.3, we consider the distribution aspects from the point of view of resource consumption. Here our main concern is to be able to infer the resource consumption distributed among the locations of the system rather than producing a monolithic expression that amalgamates the whole cost. For this purpose, we present the notion of *cost centers* and describe an underlying analysis to obtain them. In Section 3.4, we consider the inference of the cost in the presence of tasks with concurrent interleavings. This is challenging because the global variables can be modified between the time a task suspends until it resumes, and this can affect its resource consumption (e.g., the size of the data structure that a loop traverses can be increased during its suspension). We sketch a novel technique to infer the resource consumption in these cases.

- *New notions.* In this context of distributed systems, new notions of cost arise. In first place, there are new cost models that can be considered to estimate the performance of a distributed system, namely it is particularly interesting to predict the load balance of the distributed locations, the amount of data transferred among them and the parallelism achieved. Moreover, it is relevant to obtain the *peak* of the resource usage of each distributed location rather than the total amount of resources allocated in it. In order to infer such peak cost, one needs first to estimate the *queue configuration* of each distributed location, i.e., the tasks that might be simultaneously in such location queue and then we can accumulate their resource consumption together. This notion of peak is especially relevant in the context of *non-cumulative* resources that might increase and decrease along the execution. Finally, we introduce the notion of *parallel cost* which aims at overviewing the resource consumption of the overall distributed system by exploiting the parallelism among their nodes such that when tasks execute in parallel we only consider the duration of the longest among them.

- *Conclusions.* Finally, in Section 5 we conclude and point out open problems in this setting and our directions for future research.

2 Resource Analysis of Sequential Code

In this section we consider a sequential language which is deliberately simple to describe the analysis in a clear way. Distributed/concurrent operations are introduced later in Section 3. A program is a collection of methods of the form $T\ m(\textsf{int}\ x_1, \ldots, \textsf{int}\ x_k)\{s_1; s_2; \ldots, s_n;\}$, where x_i, $1 \leq i \leq k$, denote variables names and $T \in \{\textsf{int}, \textsf{void}\}$. Each instruction $s_i \in Instr$, $1 \leq i \leq n$, adheres to the following grammar:

$$s ::= x = e \mid \textsf{if}\ b\ \textsf{then}\ s\ \textsf{else}\ s \mid \textsf{while}\ b\ \textsf{do}\ s \mid x = m(\bar{y}) \mid \textsf{return}\ x$$

where x, y denote variables names. For the sake of generality, the syntax of expressions e and Boolean conditions b is not specified. As notation, for any entity A, we use $\bar{A}$ as a shorthand for $A_1, \ldots, A_n$.

A common way to rigorously represent an execution is by means of a state transition system, which is an abstract machine that consists of a set Σ of states and a binary relation $\leadsto \subseteq \Sigma \times \Sigma$, which represents transitions between states. An execution $\mathcal{E}$ starts from an initial state $\mathcal{S}_0$ containing a method call. We use $\mathcal{S}_i \leadsto_s \mathcal{S}_j$, with $\mathcal{S}_i, \mathcal{S}_j \in \Sigma$, to denote that there is a transition from $\mathcal{S}_i$ to $\mathcal{S}_j$ in which instruction s has been executed. A state is *final* iff it has no successors. Similarly, an execution is final if it finishes in a final state. Note that for our sequential language, executions consist of only one branch. However, as we will see in Section 3, for distributed and concurrent languages, multiple results for an initial call can be computed.

2.1 Cost Models

The notion of cost model for a program specifies how the resource consumption of a program is calculated, given a resource of interest. It basically defines how to measure the resource consumption, i.e., the cost, associated to each execution step and, by extension, to an entire execution. Thus, a *cost model* $\mathcal{M}$ is a function defined as $\mathcal{M} : Instr \mapsto \mathbb{R}$ and the *cost of an execution step* is defined as $\mathcal{M}(\mathcal{S} \leadsto_s \mathcal{S}') = \mathcal{M}(s)$. For instance, a cost model which counts the number of execution steps can be defined as $\mathcal{M}_{\textsf{ninst}}(s) = 1$ for any $s \in Instr$ and a cost model counting the number of times that a concrete method m is executed can be defined as:

$$\mathcal{M}_{\textsf{calls}}(s) = \begin{cases} 1 \text{ if } s \text{ is a call } m(\bar{x}) \\ 0 \text{ otherwise} \end{cases}$$

Now, given a cost model $\mathcal{M}$ and an execution $\mathcal{E}$, the *cost* of $\mathcal{E}$ w.r.t. $\mathcal{M}$, denoted as $Cost(\mathcal{E}, \mathcal{M})$ is defined as the sum of the costs of all execution steps in $\mathcal{E}$.

2.2 Upper Bounds

An upper bound for $m(\bar{x})$ w.r.t. a cost model $\mathcal{M}$, is a function $f(\bar{x}) = \textsf{cexp}$ on $\bar{x}$ which guarantees that for all $\bar{u} \in \mathbb{Z}$, and for any final execution $\mathcal{E}$ starting from

$m(\bar{u})$ it holds that $Cost(\mathcal{E}, \mathcal{M}) \leq f(\bar{u})$. The *cost expressions* cexp that can be handled in our framework follow the grammar below:

$$\mathsf{cexp} ::= r \mid \mathsf{nat}(l) \mid \mathsf{cexp}\ op\ \mathsf{cexp} \mid \log_n(\mathsf{nat}(l)+1) \mid n^{\mathsf{nat}(l)} \mid \max(S)$$

where $op \in \{+, *\}$, $r \in \mathbb{R}^+$, $n > 1 \in \mathbb{Z}^+$, $\mathsf{nat}(l)$ is defined as $\mathsf{nat}(l) = \max(\{l, 0\})$, $\max(S)$ stands for the maximum of the set of cost expressions S and l denotes a *linear expression* of the form $u_0 + u_1x_1 + \ldots u_nx_n$. The use of the nat-operator ensures that cost expressions are always evaluated to non-negative values. For instance the expression $\mathsf{nat}(x-1)$ is a valid cost expression which returns 0 for all $x \leq 1$.

The cost analysis framework that we follow [3] is based on transforming the original program in a set of cost equations by applying different static analyses and transformations on the source program. In particular, the main two steps to produce cost equations are: the transformation of the program into direct recursive form, and a size analysis which infers how the sizes of data change along the execution. From the equations, the upper bound is computed by (1) bounding the number of iterations of each recursive equation using linear ranking functions [21] and (2) by maximizing the local cost of each equation. As an example, consider the cost model which counts the number of executed instructions together with the program:

```
int m(int x, int y) {
  int r = 0, a;
  while (x < y) {
    a = p(x);
    r = r + a;
    x = x + 1;
  }
  return r;
}
```

```
int p(int x) {
  x = x + 1;
  return x;
}
```

Considering that the cost of x = x+1 is 2 (the addition plus the assignment), an upper bound for p is 3. For the case of m, first we bound the number of iterations in the while loop by means of the linear ranking function $\mathsf{nat}(y-x)$. Secondly, we multiply the bound on the number of iterations by the cost inside the loop (8, which results from the 4 instructions in the loop, 1 method call, and the 3 instructions of the method) and the cost of executing the condition (1). Thus $\mathsf{nat}(y-x)*9$ is an upper bound for the while loop. Finally, we add 3 due to the costs of the instructions outside the loop and the final evaluation of the guard, and the upper bound for m results in $m^+(x,y) = 3 + \mathsf{nat}(y-x)*9$.

Suppose now that method p has an upper bound $p^+(x) = \mathsf{nat}(x)$. Then the cost of the instruction a = p(x) is obtained by maximizing $p^+(x)$ in the context of its execution, namely x < y, which results in $\mathsf{nat}(y)$. Hence now the upper bound for m would be $m^+(x,y) = 3 + \mathsf{nat}(y-x)*(\mathsf{nat}(y)+6)$.

2.3 Extensions of Sequential Resource Analysis

The language we have used along this section does not contain a global memory, instead all variables in a method are local to it. In the presence of global variables, the computation of upper bounds becomes harder since when bounding the number of iterations of a loop we must take into account if the condition of the loop depends on a shared variable. For example, suppose we extend the language in Section 2 to support classes and objects in the standard way, where a class may contain integer fields shared by all objects of the class. Consider the following implementation of method m:

```
int m(A o1, A o2, int y) {
  int r = 0;
  while (o1.x < y) {
    a = p(o2);
    r = r + a;
    o1.x = o1.x + 1;
  }
  return r;
}

int p(A o2) {
  // read and write field o2.x
  return o2.x;
}
```

where o_1, o_2 are objects of a class A which contains a field x. The termination of the while loop depends clearly on the call $p(o_2)$ in the following sense: If o_1 and o_2 points to the same memory location, then field x is always accessed by the same reference, say o_1, and it can be treated as a local variable, what allows to apply the same techniques than in Section 2.2 in order to compute an upper bound. Otherwise, we will not be able to infer the cost as it will depend on the calling context of method m. Our approach [2] consists in computing the sequence of (*access path*) used to access each field in the program. Then, if the field is not written or its written by a unique access path, such a field is considered as *trackable*, i.e., the field can be treated as a local variable for the method. For our example at hand, it holds that in method m, the field x is read and written by two different references, o_1 and o_2 and hence the field is not trackable and the termination of the loop can not be proven. However suppose that, after the instruction int r = 0, we add $o_1 = o_2$. Now field x is written only using o_1 and thus the field is considered trackable, what allows us to compute an upper bound for method m similarly as done in Section 2.2 but in terms of $o_1.x$. More sophisticated approaches to deal with shared memory can be found in [4] and [5], where reference fields and array fields are also considered.

Another extension to sequential resource analysis is the inference of non-cumulative resources [9]. Existing cost analysis frameworks have been defined for cumulative resources which keep on increasing along the computation. In contrast, 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.

It is recognized that non-cumulative resources introduce new challenges in resource analysis [12,18]. 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. The analysis of non-cumulative resources is defined in two steps: (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).

3 Resource Analysis of Distributed Concurrent Systems

This section describes the basic extensions to resource analysis of distributed and concurrent systems.

3.1 The Language

We consider a distributed concurrent programming model with explicit locations and cooperative concurrency between the tasks at each location. Each location represents a processor with a procedure stack and an unordered buffer of pending tasks. Initially all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the buffers of any processor, including its own, or synchronize with the reception of other tasks. When a task completes, its processor becomes idle again, chooses the next pending task, and so on. The number of locations need not be known a priory (e.g., locations may be virtual). Syntactically, a location will therefore be similar to an *object* and can be dynamically created using the instruction **newLoc**. The new set of instructions of the language, extended with distributed operations from that of Section 2, is as follows:

$$s' ::= s \mid x = \textbf{newLoc} \mid x = \textbf{newDC} \mid f = x.m(\bar{y}) \mid \textbf{await}\ f? \mid x = f.\textbf{get}$$

Let us observe that now variables can hold locations and therefore the set of types is extended to $\{\textbf{void}, \textbf{int}, \textbf{loc}\}$, being **loc** the set of locations and distributed components. The special location identifier *this* denotes the current location. We can achieve different ways of distributing an application by creating new locations with **newLoc** or new distributed components by means of **newDC**. When we use **newDC**, a new distributed component is created, whereas when

we use **newLoc**, the created location (and its resource consumption) belongs to the current distributed component.

The language is also extended with *future variables*, denoted by f in the grammar, which are used to check if the execution of an asynchronous task has finished. Method calls on locations are asynchronous and are associated with a future variable that will hold their result. The instruction **await** $f?$ allows synchronizing the execution of the current task with the task which the future variable f is pointing to; and instruction $x = f.\mathbf{get}$ is used to retrieve the value stored in f.

3.2 Cost Models

In Section 2.1 we presented some important cost models for sequential programs. However, other interesting cost models can be defined in distributed and concurrent systems, as shown in [1]. For instance, a cost model that counts the total *number of distributed components* (*number of locations*), created along the execution can be defined as $\mathcal{M}_{\mathsf{loc}}(s) = 1$ if $s \equiv x = \mathbf{newDC}$ (**newLoc**) and $\mathcal{M}_{\mathsf{loc}}(s) = 0$ otherwise. Since distributed components are the distribution units, this cost model provides an indication on the amount of parallelism that might be achieved.

A cost model that counts instructions of the form $x.m(\bar{y})$ can be used to infer the *number of tasks* that are spawned along an execution. This cost model can be refined to count the number of calls to specific methods, locations or distributed components by focusing on specific method and object names.

Communications play a fundamental role in the design of a distributed system, because they influence their performance. A cost model that counts the *number of communications* or the amount of *transmitted data* is very useful when designing distributed systems. The goal of such cost models is to infer, not only the number of communications between locations or distributed components, but also the sizes of the arguments in the task invocation and of the returned values. This cost model that over-approximates the amount of data transmitted uses size analysis [13] to infer upper bounds on the data sizes at the points in which tasks are spawned. In particular, given an instruction $x.m(\bar{y})$ it over-approximates the size of $\bar{y}$ and also of the returned value.

3.3 Distribution: Cost Centers

In a distributed setting, the above notion of cost model has to be extended because, rather than considering a single component in which all steps are performed, we have in general multiple locations and distributed components possibly running concurrently and/or distributively on different CPUs. Thus, rather than aggregating the cost of all executing steps, it is required to treat execution steps which occur on different locations or components separately. With this aim, we adopt the notion of *cost centers* [20], proposed for profiling functional programs. The upper bounds will use cost centers in order to keep the resource usage assigned to the different components separate.

Ideally, one would like to have a different cost center for each different location or distributed component created along the execution of the program. However this cannot be determined statically and has to be approximated. For this aim, we rely on points-to analysis in order to approximate the set of locations or distributed components which each reference variable may point to during program execution. This allows us to make the analysis object-sensitive and separate the cost that corresponds to different instances of locations and/or distributed components that are created at the same program point but that correspond to different object names and may belong to different distributed components.

3.4 Concurrency: MHP-based Analysis

Resource consumption inference in concurrent and distributed systems is more difficult than in the sequential case, since different tasks can interleave their executions and therefore change the value of shared variables. This situation becomes clearer in the following example from [11], where g is a shared variable and x is a variable local to S_2:

S_1
```
1 while (g > 0){
2   g = g − 1;
3   await *?
4 }
```

S_2
```
5 while (x > 0){
6   x = x −1;
7   g = *;
8 }
```

The instruction at L7, that updates the field g, may interleave with **await** *? at L3 in S_1. Therefore the number of iterations of the loop S_1 may differ from the original value of g, as the value of that shared variable can change between iterations. To infer the number of iterations of S_1 we use the following approach [11]:

1. Locate those instructions that update shared variables and can interleave with the loop. In the example, the only interleaving instruction that updates g is L7. To obtain this information we use a *may-happen-in-parallel* analysis [10]. This analysis over-approximates the pairs of program instructions that can execute in parallel or in an interleaved way.
2. Find an upper-bound on the number of times that those interleaving instructions are executed. This computation may require the recursive calculation of upper bounds for other loops. In the example above, a sound and precise bound on the number of executions of L7 is x, since x is a local variable.
3. Finally, the upper bound for S_1 is the maximum number of iterations ignoring the instruction **await** *? , but assuming that at this point g can take its maximum value g^+, multiplied by the maximum number of visits to L7. Thus, $g^+ * x$ is a sound upper bound.

Once we have computed an upper bound on the iterations of the loop, we can easily infer the concrete resource consumption by using a concrete cost model. Notice that the may-happen-in-parallel analysis is crucial, since it will be able

to discard some spurious interleavings that will lead to imprecise upper bounds. Otherwise, we will be forced to consider that every updating instruction could interleave with every loop. Note also that the may-happen-in-parallel analysis is independent and it is used as a *black-box*, so any improvement on it will enhance the upper bounds automatically.

4 New Notions of Cost in Distributed Systems

Building upon the basic analysis presented in the previous section, in this section we describe new cost models and notions of cost that appear in distributed systems.

4.1 Advanced Cost Models

By building upon the cost models described in Section 3.2, we have defined several advanced cost models that provide indicators to assess the level of distribution in the system [7], the amount of communication among distributed nodes that it requires, and how balanced the load of the distributed nodes that compose the system is. Our indicators are given as functions on the input data sizes, and they can be used to automate the comparison of different distributed settings and guide towards finding the optimal configuration. Let us see an example to explain these issues:

```
1 void m(int n){
2   loc a = newLoc | newDC;
3   while (n > 0) {
4     loc b = newLoc | newDC;
5     b.p(n,a);
6     n = n − 1;
7   }
8 }
```

```
9 void p(int n,loc a) {
10   while (n > 0) {
11     a.q();
12     n = n − 1;
13   }
14 }
15 q () { 10 instr }
```

Method m creates one location using **newLoc**(or distributed component using **newDC**) at L2 pointed by variable a and it contains a loop that creates n locations (or distributed components) at L4. Such loop also spawns n tasks executing method p (L5). Method p contains a loop that calls q n times (L11). Those program points where locations are created, L2 and L4, are crucial for determining the behaviour of the system. Depending on the creation of a distributed component (**newDC**) or a location (**newLoc**), we obtain a different setting whose performance could be radically different from the others. To evaluate which setting has a better performance, we define the notion of *performance indicator*. A performance indicator is a function, expressed in terms of the input arguments of the program, that evaluates to a number in the range [0–1], such that the closer to one the better the performance. We define three different indicators:

1. The *distribution function* ($\mathcal{D}$) measures how much distributed the application is. It is defined as the relation between the number of distributed components that are created for this particular setting with respect to the maximum number of potential distributed components that could be created if all location instances were distributed components, i.e., the optimal setting from a distribution perspective in which we have as many distributed components as possible.
2. The *communication function* ($\mathcal{K}$) aims at measuring the level of external communications performed (i.e., calls to locations that belong to other distributed components). The motivation is that calls to other distributed components are potentially more expensive (as they require communications costs) and thus one wants to minimize them as much as possible. It is defined as one minus the ratio between the number of communications that the program performs in the current setting, and the maximum number of communications when using a setting in which all locations are created as distributed components and thus every asynchronous call (on a location different from the one executing) is external.
3. The *balance function* ($\mathcal{B}$) measures the balance level of the distributed system. We consider that the system is optimally balanced when all its components execute the same number of instructions. The *balance function* makes use of the upper bounds on the number of instructions and the upper bounds on the number of distributed components (and locations) to measure the standard deviation of the number of instructions executed by each distributed component. As we want to measure the balance level by means of a number in the interval [0–1] as in the other indicators, we divide the standard deviation by the maximum dispersion of the distributed components from the average.

Figure 1 shows the graphical representation of the functions $\mathcal{D}$, $\mathcal{K}$ and $\mathcal{B}$ for two possible settings by using **newLoc** or **newDC** at L2 and L4 of the program shown above. By means of the evaluation of the performance indicators we can observe that for Setting 1 higher values of n lead to a better distribution behaviour because a new distributed component is created at each iteration of the loop in m. Regarding communications, Setting 2 behaves better for lower values of n, but for higher values of n, both settings behave badly (close to 0). In addition, the evaluation of the balance function indicates that the load of the system is better balanced with Setting 1. The information obtained from the performance indicators could be extremely useful in the deployment process of a distributed system. In order to find the optimal setting for a distributed system, we should be able to: (1) generate all possible settings automatically, (2) generate performance indicators for each of them and (3) be able to compare such indicators for the different settings.

4.2 Peak Cost

The framework presented so far allows us to infer the total number of instructions that it needs to execute, the total amount of memory that it will need

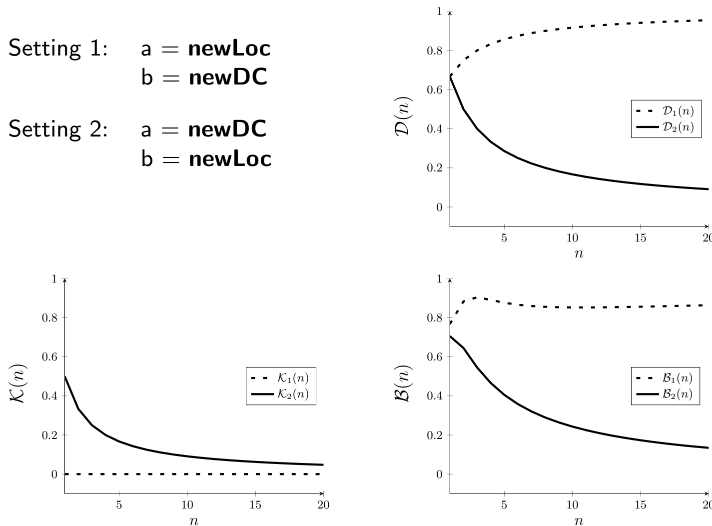

Fig. 1. Graphical representation of the functions $\mathcal{D}$, $\mathcal{K}$ and $\mathcal{B}$

to allocate, or the total number of tasks that will be added to its queue. This is a too pessimistic estimation of the amount of resources actually required in the real execution. The amount of work that each location has to perform can greatly vary along the execution depending on: (1) the amount of tasks posted to its queue, (2) their respective costs, and (3) the fact that they may be posted in parallel and thus be pending to execute *simultaneously*. In order to obtain a more accurate measure of the resources required by a location, the *peak* of the resource consumption can be inferred instead [8], which captures the maximum amount of resources that the location might require along any execution. In addition to its application to verification, this information is crucial to dimensioning the distributed system: it will allow us to determine the size of each location *task queue*; the required size of the location's memory; and the processor execution speed required to execute the peak of instructions and provide a certain response time. It is also of great relevance in the context of software *virtualization* as used in cloud computing, as the peak cost allows estimating how much processing/storage capacity one needs to buy in the host machine, and thus can greatly reduce costs.

Inferring the peak cost is challenging because it increases and decreases along the execution, unlike the standard notion of *total cost* which is cumulative. To this end, it is very relevant to infer, for each distributed component, its *abstract queue configuration*, which captures all possible configurations that its queue can take along the execution. A particular queue configuration is given as the sets of tasks that the location may have pending to execute at a moment of time. For

instance, let us see the following example program, which has as entry method ex1:

```
1 void ex1() {           6 void m1() {          12 void m2() {
2   ff = this.m1();      7   fa = x.a();        13   x.d();
3   await ff?;           8   await fa?;         14   x.e();
4   this.m2();           9   fb = x.b();        15 }
5 }                      10  await fb?;
                         11 }
```

It first invokes method m1, which spawns tasks a and b. Method m1 guarantees that a and b are completed when it finishes. Besides, we know that the **await** instruction in L8 ensures that a and b cannot happen in parallel. Method m2 spawns tasks d and e and does not await for their termination. We can observe that the **await** instructions in m1 guarantee that the queue is empty before launching m2. We can represent the tasks in the queue of location x by the tasks queue graph by means of the following queue configurations: $\{\{a\}, \{b\}, \{d, e\}\}$.

In order to quantify queue configurations and obtain the peak cost, we need to over-approximate: (1) the number of instances that we might have running simultaneously for each task and (2) the worst-case cost of such instances. The main extension is to define cost centers of the form $c(o{:}m)$ which contain the location name o and the task m running on it. Now, using the upper bounds on the total cost in Section 3.3 we already gather both types of information. This is because the cost attached to the cost center $c(o{:}m)$ accounts for the accumulation of the resource consumption of *all* tasks running method m at location o. We therefore can safely use the total cost of the entry method $p(\bar{x})$ restricted to $o{:}m$, denoted $p^{+}(\bar{x})|_{\{o:m\}}$, as the upper bound of the cost associated with the execution of method m at location o which sets up to 0 the cost centers different from $c(o{:}m)$. The key idea to infer the *quantified queue configuration*, or simply *peak cost*, of each location is to compute the total cost for each element in the set of abstract configurations and stay with the maximum of all of them. In the previous example, the peak cost of location x in ex1 is $max\{ex_1^{+}(n)|_{c_1}, ex_1^{+}(n)|_{c_2}, ex_1^{+}(n)|_{c_3}\}$, where $c_1 = \{x{:}a\}$, $c_2 = \{x{:}b\}$ and $c_3 = \{x{:}d, x{:}e\}$.

4.3 Parallel Cost

Parallel cost differs from the standard notion of *serial cost* by exploiting the truly concurrent execution model of distributed processing to capture the cost of synchronized tasks executing in parallel. It is also different to the peak cost since this one is still serial; i.e., it accumulates the resource consumption in each component and does not exploit the overall parallelism as it is required for inferring the parallel cost. It is challenging to infer parallel cost because one needs to soundly infer the parallelism between tasks while accounting for waiting and idle processor times at the different locations. We are currently developing a static analysis to obtain the parallel cost.

5 Conclusions and Future Research

Inferring the resource consumption (a.k.a cost) of computer programs, which is a general form of complexity, is one of the most fundamental tasks in computer science, and its automation has been the subject of voluminous research in the last decade.

Research in this area resulted in several cost analysis frameworks for sequential low- and high-level modern programming languages, such as the sequential fragments of Java and its corresponding low-level bytecode. These frameworks have been enhanced overtime to scale for large programs, and to handle programs with complex control-flow and sophisticated heap data-structures. They have also been extended to support non-cumulative cost models in which resources can be released as well, e.g., memory consumption in the presence of garbage collection. The underlying complexity analyses employed by these frameworks range from the classical worst/best case approach to more advanced ones such as the amortised analysis approach, and thus they offer users a wide range of performance/precision trade-offs. Some of these frameworks also provide support for certification and verification of resource consumption.

Research in recent years has concentrated on extending the sequential cost analysis frameworks to handle concurrency and distribution. The main challenge was to handle new notions of cost that are more suitable for such programming paradigms. This includes the peak cost, that refers to the maximal amount of resources that can be used simultaneously (by different tasks), and the parallel cost, that do not accumulate the cost of tasks that are executing in parallel on different computing units. The underlying techniques for these notions of cost rely on the use of MHP analysis, which provides information on which tasks might interleave or execute in parallel. Another important functionality that was introduced is the ability to attribute cost to particular nodes of a distributed system, which is of utmost importance for optimizing the resource usage of such systems or balancing the load of their nodes.

In spite of the remarkable achievements in the field of cost analysis, there are still several directions that need to be considered in the future: (1) exploring new applications for cost analysis. A promising direction is the use of cost analysis to identify security vulnerabilities that are related to resource consumption; (2) current techniques for cost analysis of concurrent programs predict the cost at the algorithmic level, more work is required to leverage these techniques to take the underlying (multi-core) architecture into account. This would require supporting more sophisticated concurrency models; (3) in the context of parallelism, cost analysis of massive parallel programs has not been investigated yet, more attentions should be paid to such programming paradigms as they are popular in scientific communities; (4) support for probabilistic information is probably the most important and appealing direction. Probabilistic distributions can be used to describe a cost model, which allows constructing platform dependent cost models (e.g., energy) using profiling tools. Probabilistic distributions can be also used to describe the distribution of the input data, which can then be used to infer notions such average cost and distribution of cost.

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 MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006.

References

1. E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *Proc. of FM'09*, volume 5850 of *LNCS*, pages 370–386. Springer, 2009.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Ramírez. From Object Fields to Local Variables: A Practical Approach to Field-Sensitive Analysis. In *Proc. of SAS'10*, volume 6337 of *LNCS*, pages 100–116. Springer, 2010.
5. E. Albert, P. Arenas, S. Genaim, G. Puebla, and G. Román-Díez. Conditional Termination of Loops over Heap-allocated Data. *Science of Computer Programming*, 92:2 – 24, 2014.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413(1):142–159, 2012.
7. E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Quantified Abstract Configurations of Distributed Systems. *Formal Aspects of Computing*, 2015. http://dx.doi.org/10.1007/s00165-014-0321-z.
8. E. Albert, J. Correas, and G. Román-Díez. Peak Cost Analysis of Distributed Systems. In *Proc. of SAS'14*, volume 8723 of *LNCS*, pages 18–33, 2014.
9. E. Albert, J. Correas, and G. Román-Díez. Non-Cumulative Resource Analysis. In *Procs. of TACAS'15*, volume 9035 of *LNCS*, pages 85–100. Springer, 2015.
10. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
11. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*, LNCS 8172, pages 349–364. Springer, October 2013.
12. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM, 2010.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.
14. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
15. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
16. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.

17. J. Hoffmann and Z. Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *Functional and Logic Programming - 12th International Symposium, FLOPS*, volume 8475 of *LNCS*, pages 152–168. Springer, 2014.
18. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of POPL'13*, pages 185–197. ACM, 2003.
19. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
20. Richard G. Morgan and Stephen A. Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programing*, 8(3):201–237, 1998.
21. A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, volume 2937 of *LNCS*, pages 239–251, 2004.
22. D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.
23. B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.