Resource Analysis of Distributed Systems

Elvira Albert[1](✉), Jesús Correas[1], and Guillermo Román-Díez[2]

[1] DSIC, Complutense University of Madrid, Madrid, Spain
elvira@sip.ucm.es
[2] DLSIIS, Technical University of Madrid, Madrid, Spain

Abstract. Distributed systems are composed of nodes that communicate and coordinate their actions by passing messages. The nodes interact with each other in order to achieve a common goal. Resource analysis of distributed systems needs to consider the distribution, communication and interaction aspects of the systems as well. We sketch the basic framework proposed for the resource analysis of distributed systems, together with the new notions of cost that arise in such distributed context. In particular, we will discuss the notions of: *peak cost* that captures the maximum amount of resources that each distributed node might require along the whole execution; and *parallel cost* which corresponds to the maximum cost of the execution by taking into account that, when distributed tasks run in parallel, we need to account only for the cost of the most expensive one. The framework is developed for a concurrent objects language with futures, a formalism that is based on Frank's work.

1 Introduction

Static resource analysis [18] aims at inferring an *upper bound* on the amount of resources required along any execution of a software system by only inspecting its code and without executing it [3,11,12,19]. We rely on a *generic* resource analysis framework [2,3] that is parametric w.r.t. the type of resource that one wants to measure. Traditional resources include the number of steps executed, the amount of memory allocated, or the number of calls to a specific method.

Distributed systems pose new challenges to resource analysis [17]. The fact that they are composed of a number of distributed nodes that communicate by exchanging messages needs to be considered by the analysis. We consider a simple class-based programming language with four instructions to define the distributed execution model: (1) new C creates a new distributed component, referred to as a *location*, that executes methods of class C, (2) f=a.m($\overline{x}$) spawns an asynchronous task m($\overline{x}$) on the location a, and f is a future variable that allows us to check whether the asynchronous task has been completed, (3) the instruction await f? allows us to synchronize with the termination of the task associated to the future variable f, and (4) the instruction f.get returns the value computed by the task associated to the future variable f (or blocks if the task has not terminated yet). We omit class definitions when they are not relevant

© Springer International Publishing Switzerland 2016
E. Ábrahám et al. (Eds.): de Boer Festschrift, LNCS 9660, pp. 33–46, 2016.
DOI: 10.1007/978-3-319-30734-3_5

for the examples. This language is the core of ABS [14], a concurrent objects language with *futures*. A formal semantics for such language can be found in Frank's work [10].

The notion of *cost center* [1] is fundamental to define the framework of resource analysis for distributed systems. The main idea is that it allows splitting the cost of executing the whole system at the granularity of interest. For instance, one can observe the cost associated to each distributed component. And it allows observing the cost associated to executing a certain task within a distributed component. Using cost centers, we can define new performance indicators for distributed systems [6]. Consider the distributed system depicted in Fig. 1 which is composed of four distributed components. Our interest is in inferring performance indicators that allow us to estimate the overall performance of the distributed system. One of the main indicators will be the one that determines whether the load is well balanced among the distributed nodes. For this purpose, we infer the resource usage for each of the distributed nodes (in the figure it appears in blue over the node). Note that since the computation depends on input variables n and m, the resource usage is given by means of cost expressions that can be evaluated for concrete input values for n and m. By comparing such cost expressions, we can identify whether there is a bottle neck in the system (for instance the resource usage of the upper component is exponential and this might be too expensive). Another essential performance indicator is the one that estimates the sizes of the communication among the distributed components. This is depicted in the figure by arrows whose labels indicate the amount of data sent from one component to another. Again, since this might depend on the input data, it is expressed by means of cost expressions in terms of the input values. This way we are able to approximate communication costs.

Besides defining new performance indicators, there are new notions of cost that arise in the context of distributed systems. In particular, we pursue the notion of *peak cost* [7] which corresponds to the maximum amount of resources that the location might require along any execution. Inferring the peak cost is not trivial, since we need to infer: (1) the amount of tasks posted to its queue, (2) their respective costs, and (3) knowledge on whether the tasks may be posted in parallel and thus be pending to execute simultaneously.

The other notion of cost that we are able to infer is the *parallel cost* [4], which differs from the standard notion of serial cost because when tasks execute in parallel it only considers the cost of the most expensive one. Thus, it is different from the standard notion of cost because it exploits 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 the peak cost is 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. The main challenge to infer the parallel cost is to infer the parallelism between tasks while accounting for waiting and idle processor times at the different locations.

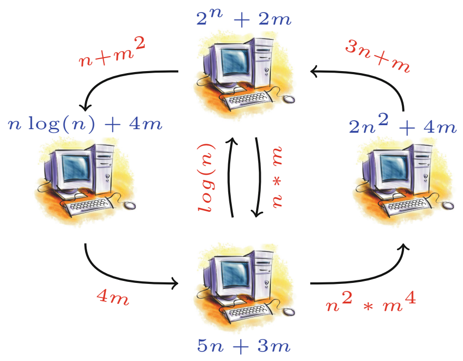


Fig. 1. Performance indicators in a distributed system

The contribution of this paper is putting within the same setting different analyses that have been published in the following venues: the analysis that underlie performance indicators is developed in [5,6], the peak cost analysis puts together work published at [7,8], and the parallel cost analysis was introduced at [4]. The rest of the paper is structured as follows: Sect. 2 describes the basics of the resource analysis framework. Section 3 introduces the indicators that can be considered to estimate the performance of a distributed system. In Sect. 4 we overview the main ideas of the peak cost analysis. Section 5 intuitively explains the notion of parallel cost and an analysis that overapproximates it. Finally, Sect. 6 concludes and points out some directions for future work.

2 Resource Analysis with Cost Centers

The notion of cost center is an artifact used to define the granularity of a cost analyzer. In [1], the proposal is to define a cost center for each distributed component; i.e., cost centers are of the form $c(o)$ where o is a location identifier and $c(\_)$ is the artifact used in the cost expressions to attribute the cost to the different components. Every time the analyzer accounts for the cost of executing an instruction *inst* at program point *pp*, it also checks at which location the instruction is executing, since the instruction might be reached from executions on different distributed components. This information can be approximated by an analysis that is called points-to, and different levels of precision can be achieved (see e.g. [15,16]). In particular, given a program point *pp* and the current distributed location *this*, points-to analysis returns the set of locations $O_{pp} = pt(pp, this)$ which along the execution *this* can be instantiated to. The cost of the instruction is accumulated in the cost centers of all elements in O_{pp} as

$$\sum_{\forall o \in O_{pp}} c(o) * cost(inst),$$

where $cost(inst)$ expresses in an abstract way the cost of executing the instruction. If we are counting steps, then $cost(inst) = 1$. If we measure time, $cost(inst)$

refers to the time to execute *inst*. Then, given a method $m(\bar{x})$, the cost analyzer will compute an upper bound for the serial cost of executing m of the form $m^{+}(\bar{x}) = \sum_{i=1}^{n} c(o_i){*}C_i$, where o_i refers to a location and C_i is a cost expression that bounds the cost of the computation carried out by location o_i when executing m. Thus, cost centers allow computing costs at the granularity level of the distributed components. If one is interested in studying the computation performed by one particular component o_j, we simply replace all $c(o_i)$ with $i \neq j$ by 0 and $c(o_j)$ by 1.

```
void m (int n) {
  ...
  x.p();
  ...
  y.q();
  ...
}

void p () {
  ...
  y.s();
  ...
}

void q () {
  ...
}
void s () {
  ...
}
```

Fig. 2. Example of resource analysis with cost centers

Example 1. For the code excerpt in Fig. 2, we have three cost centers for the three locations that accumulate the costs of the code they execute: the cost center for the location executing m, namely o, and the cost centers for the locations referenced by x and y, that we suppose already created. Therefore, we have that the cost of executing m is $m^{+}(n) = c(o){*}\widehat{m} + c(x){*}\widehat{p} + c(y){*}\widehat{s} + c(y){*}\widehat{q}$, where we represent with $\widehat{z}$ the cost of the instructions in method z. ■

Other Types of Granularity. But, besides the original idea of using the cost centers to represent the distributed components, they can be used to achieve other kinds of granularity in the analysis. In particular, in the peak cost analysis (Sect. 4), they will allow us to achieve *task-level* granularity; and in the parallel cost analysis (Sect. 5), to achieve *block-level* granularity, as explained below.

As for task-level granularity, one wants to obtain the cost associated to the execution of each task m when executed on each distributed component o. To this purpose, we define cost centers of the form $c(o{:}m)$ which contain the location identifier o and the task m running on it. Then, every time the analyzer accounts for the cost of executing an instruction *inst*, it checks at which location *inst* is executing (e.g., o) and to which method it belongs (e.g., m), and accumulates $c(o{:}m) * cost(inst)$. As for the block-level granularity, we define block-level cost centers $c(o{:}b)$ which contain the location identifier o and the block b running on it.

Let $\mathcal{M}$ be a set that contains all method names combined with all location identifiers where they can be executed. Given a method $m(\bar{x})$, the cost analyzer now computes a *task-level upper bound* for the cost of executing m. This upper bound is of the form $m^{+}(\bar{x}) = \sum_{i=1}^{n} c(o_i{:}m_i) * C_i$, where $o_i{:}m_i \in \mathcal{M}$, and C_i is a cost expression that bounds the cost of the computation carried out by location o_i while executing block m_i. Let $\overline{\mathcal{B}}$ be a set that contains all blocks combined with all location identifiers where they can be executed. Given a method $m(\bar{x})$, the cost analyzer now computes a *block-level upper bound* for the cost of executing m. This upper bound is of the form $m^+(\bar{x}) = \sum_{i=1}^{n} c(o_i{:}b_i) * C_i$, where $o_i{:}b_i \in \overline{\mathcal{B}}$, and C_i is a cost expression that bounds the cost of the computation carried out by location o_i while executing block b_i. Observe that b_i need not be a block of m because we can have transitive calls from m to other methods; the cost of executing these calls accumulates in $m^+(\bar{x})$.

As we have seen, resource analysis allows different levels of granularity, thus we can have different types of cost center artifacts. For any kind of granularity, the notation $m^+(\bar{x})|_{cc}$ is used to express the cost associated to the cost center $c(cc)$ within the cost expression $m^+(\bar{x})$, i.e., the cost obtained by setting all $c(cc_i)$ to 1 when $cc' = cc$ and to 0 otherwise. Given a set of cost centers $N = \{cc_1, \ldots, cc_n\}$, we let $m^+(\bar{x})|_N$ refer to the cost obtained by setting to one the cost centers $c(cc_i)$ such that $cc_i \in N$.

3 Performance Indicators

In this section we define indicators that can be considered to estimate the performance of a distributed system [6]. In particular, we are interested in predicting the load balance of the distributed locations, the number of communications between nodes and the amount of data transferred among them.

3.1 Load Balance

Using the cost centers described in Sect. 2, we define an indicator to assess how balanced the load of the distributed nodes that compose the system is. By attributing the cost of each instruction to the location responsible of executing it, upper bounds can help during the development process to take better design decisions for obtaining an optimal load balancing.

Example 2. In the source code shown in Fig. 3(left and center), method m creates a new location using new at L2, pointed by variable a, and then a while loop spawns n tasks executing method p (L4). Besides, method p contains another loop that calls q n times (L10). Observe that the second argument of the call to p at L4 causes method q to be executed at location a. If we replace the second argument by this at L4, that is a.p(n,this), method q will be executed at the location executing m. We refer to this location as o. The upper bound expressions for the number of steps are the same for both cases, but such decision is crucial for properly balancing the system. By using the resource analysis of Sect. 2, for a.p(n,a) at L4, the cost attributed to o is $m^+(n)|_o = 9 + 7 * n$ and the cost attributed to a is $m^+(n)|_{\mathsf{a}} = 1 + n * (6 + 14 * n)$. It can be seen that the program is not properly balanced, since $m^+(n)|_o$ is a linear expression w.r.t. the value of n, while $m^+(n)|_{\mathsf{a}}$ is a quadratic expression. On the other hand,

```
1 void m (int n) {
2   loc a = new Obj();
3   while (n > 0) {
4     a.p(n,a);
5     n = n − 1;
6   }
7 }
```

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

```
15 void m2 (int n) {
16   while (n > 0) {
17     loc a = new Obj();
18     a.p(n,a);
19     n = n − 1;
20   }
21 }
```

Fig. 3. Example of performance indicators

by using a.p(n,this) at L4, we have that $m^+(n)|_{\mathsf{a}} = 1 + n * (6 + 7 * n)$ and $m^+(n)|_o = 9 + n * (7 + 7 * n)$. In this case we can see that the program is more evenly balanced, as both expressions are quadratic w.r.t. n.

When reasoning about distributed systems, it is essential to have information about their configuration, i.e., the sorts and quantities of nodes that compose the system. As we have seen in the previous example, configurations may be straightforward in simple applications, but the tendency is to have rather complex and dynamically changing configurations (cloud computing is an example of this). To this end, in addition to the upper bound on the number of instructions executed by each location, it is required to have information about how many instances of each location might exist. Resource analysis described at Sect. 2 can also be extended to provide such information.

Example 3. As we have seen in Example 2, method m only uses two locations, o and a. In contrast, method m2 shown in Fig. 3(right) creates locations within a loop and, by means of the resource analysis, we can infer that the number of locations created at m2 is bounded by the value of n.

If we consider that a system is optimally balanced when all its components execute the same number of instructions, we can use the upper bounds on the number of instructions and the upper bounds on the number of distributed components to reason about how balanced the load of the distributed nodes that compose the system is. As regards the number of instructions executed by each location in the system, we have to take into account that an abstract location might represent multiple concrete locations. This means that the number of instructions executed by an abstract location actually accounts for the instructions executed by all locations it represents.

Example 4. The analysis of m2 returns that $m2^+(n)|_o = 6 + 10 * n$ and that the number of instructions executed by all locations created within the loop is $m2^+(n)|_{\mathsf{a}} = n * (7 + 14 * n)$. As we have seen in Example 3, the number of locations created within the loop is bounded by the value of the input argument n. Therefore, we have n locations that execute $n * (7 + 14 * n)$ instructions, and another location o that executes $6 + 10 * n$ steps, which implies that the system is properly balanced.

3.2 Number of Transmissions and Transmission Data Sizes

Knowledge of the number of communications and the transmission data sizes is essential, among other things, to predict the bandwidth required to achieve a certain response time, or conversely, to estimate the response time for a given bandwidth. The different locations of a distributed system communicate and coordinate their actions by posting tasks among them. A task is posted by building a message with the task name and the data on which such task has to be executed. When the task completes, the result can be retrieved by means of another message from which the result of the computation can be obtained. Thus, the transmission data size of a distributed system mainly depends on the amount of messages posted among the locations of the system, and the sizes of the data transferred in the messages. In order to estimate the transmission data sizes, we need to keep track of the amount of data transmitted in two ways:

1. By posting asynchronous tasks among the locations. This requires building a message in which the name of the task to execute and the data on which it executes are included.
2. By retrieving the results of executing the tasks. In our setting, future variables are used to synchronize with the completion of a task and retrieve the result.

Our analysis infers a safe over-approximation of the transmission data sizes required by both sources of communications in a distributed system. Our method infers two different pieces of information: the number of tasks spawned at a given location, and the data sizes transmitted as a result of the task spawned.

Since we are considering an abstract representation of data by means of functional types, we will focus on *units of data* transmitted instead of bits, which depends on the actual implementation and is highly platform-dependent. Concretely, we assume that the cost of transmitting a basic value or a data type constructor is one unit of data. This size measure is known as *term size*. However, our static analysis would work also with any other mapping from data types to corresponding sizes (given by means of a function α).

Example 5. The example program showed in Fig. 4 creates locations s and m at L6 and L7, respectively, to perform some processing on a list. The list l has an initial content set at L5 (not relevant for the example) that is passed as a parameter of the call to method work at location m, and thus there is data transmission at this point. Method work extends the list with n values, and calls method process at location s (L23) after adding each element to the list, passing the list as argument. Method process does some processing to the list passed as argument. There are two program points in method work where data is transmitted between locations m and s: L23 and L24, that correspond to the call to process and the retrieval of the returned value, respectively.

Data structures are defined by means of data constructs, as it is showed in L1 with the data type definition for representing lists of integers. We consider the term size of data structures as the size measure. For example, a list defined as l = Cons(1, Cons(2, Cons(3,Nil))) has size $\alpha(\mathsf{l}) = 7$, as it counts 2 for each

```
 1 data List = Nil | Cons(Int, List);
 2 // main method
 3 Unit main (Int n) {
 4  Slave s; Master m; List l;
 5  l = ...;
 6  s = new Slave();
 7  m = new Master(s);
 8  m.work(l,n);
 9 }
10 class Slave {
11  Int process (List le) {
12   ...
13   return h;
14 }}// end class
15 class Master {
16  Slave s;
17  work(List l, Int n) {
18   Int x;
19   Int n;
20   fut<Int> y;
21   while (n>0){
22    l = Cons(n,l);
23    y = s.process(l);
24    x = y.get;
25    n−−;
26   }
27  }
28 }// end class
```

Fig. 4. Example of transmission data sizes

element in the list (the Cons constructor and the element itself), plus 1 for the Nil constructor.

For inferring an upper bound on the number of tasks spawned between all pairs of distributed locations, we use the cost analysis framework described in Sect. 2. In particular, we need to use a *symbolic* cost center that allows us to annotate the caller and callee locations when a task is spawned in the program. In essence, if we find an instruction a.m(x) which spawns a task m at location a, the cost model symbolically counts $c(this, a, m) * 1$, i.e., it counts that 1 task executing m is spawned from the current location this at a. If the task is spawned within a loop that performs n iterations, the analysis will infer $c(this, a, m) * n$.

Example 6. For the code in Fig. 4, cost analysis infers that the number of iterations of the loop in work (at L21) is bounded by the expression $\mathsf{nat}(n)$. Function $\mathsf{nat}(x) = max(x, 0)$ is used to avoid negative evaluations of the cost expressions. Then, by applying the number of tasks cost model we obtain the following expression that bounds the number of tasks spawned at L23: $c(\mathsf{m}, \mathsf{s}, \mathsf{process}) * \mathsf{nat}(n)$.

The second piece of information obtained by our analysis is the data sizes transmitted as a result of spawning a task. To this end, we need to infer the sizes of the arguments in the task invocations. Typically, size analysis [9] infers upper bounds on the data sizes at the end of the program execution. Here, we are interested in inferring the sizes at the points in which tasks are spawned. In particular, given an instruction a.m(x), we aim at over-approximating the size of x when the program reaches the above instruction. If the above instruction can be executed several times, we aim at inferring the largest size of x, denoted $\alpha(x)$, in all executions of the instruction. Altogether, $c(this, a, m) * \alpha(x)$ is a safe over-approximation of the data size transmission contributed by this instruction. The analysis will infer such information for each pair of locations in the system that communicate, annotating also the task that was spawned.

Example 7. Since in method work the size of l is increased within the loop at L22, the maximum size of l is produced in the last call to process. Recall that the term size of the list l counts 2 units for each element in the list. Therefore, each iteration of the loop at L21 increments the term size of the list in 2 units and, consequently, the last call to process is done with a list of size $l_0 + 2 * n$, where l_0 is the term size of the initial list, created at L5. In addition, the value returned by the call to process is retrieved at L24. Since the data retrieved is of type Int, its size is 1. Then, the data transmitted between locations m and s is bounded by the following expression, where the constant $\mathcal{I}$ is the size of establishing the communication:

$$c(\mathsf{m}, \mathsf{s}, \mathsf{process}) * \mathsf{nat}(n) * (\mathcal{I} + \mathsf{nat}(l + n * 2)) + c(\mathsf{s}, \mathsf{m}, \mathsf{process}) * \mathsf{nat}(n) * (\mathcal{I} + 1).$$

4 Peak Cost Analysis

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 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 [7], 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:

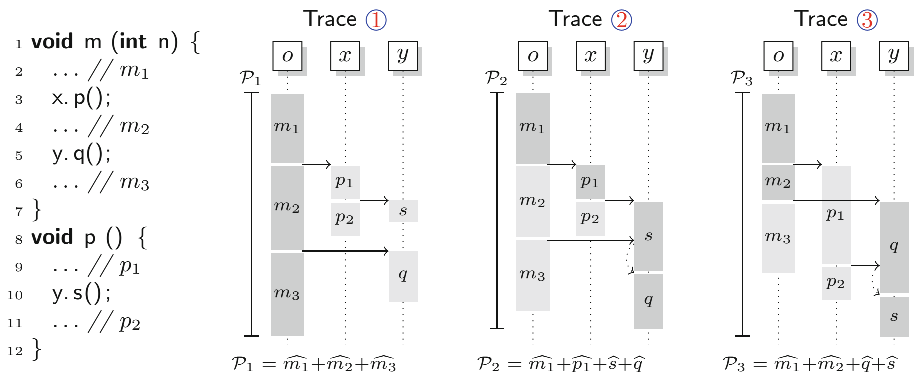
```
1 void ex1() {
2   ff = this.m1();
3   await ff?;
4   this.m2();
5 }

6 void m1() {
7   fa = x.a();
8   await fa?;
9   fb = x.b();
10  await fb?;
11 }

12 void m2() {
13  x.d();
14  x.e();
15 }
```

It first invokes method m1, which spawns tasks a and b at location x. 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 has been to define cost centers of the form $c(o{:}m)$ which contain the location identifier o and the task m running on it, as explained in Sect. 2. Now, using the upper bounds on the total cost 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 $ex1(\bar{x})$ restricted to $o{:}m$, denoted $ex1^{+}(\bar{x})|_{\{o:m\}}$, 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 $max\{ex1^{+}(n)|_{c_1},\ ex1^{+}(n)|_{c_2}, ex1^{+}(n)|_{c_3}\}$, where $c_1 = \{x{:}a\}$, $c_2 = \{x{:}b\}$ and $c_3 = \{x{:}d, x{:}e\}$.

5 Parallel Cost Analysis

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 [13]. 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. Let us see an example.



```
void m (int n) {
  ... // m1
  x.p();
  ... // m2
  y.q();
  ... // m3
}
void p () {
  ... // p1
  y.s();
  ... // p2
}
```

Trace ①
Trace ②
Trace ③
o
x
y
$\mathcal{P}_1$
$\mathcal{P}_2$
$\mathcal{P}_3$
m_1
m_2
m_3
p_1
p_2
s
q
$\mathcal{P}_1 = \widehat{m_1}+\widehat{m_2}+\widehat{m_3}$
$\mathcal{P}_2 = \widehat{m_1}+\widehat{p_1}+\widehat{s}+\widehat{q}$
$\mathcal{P}_3 = \widehat{m_1}+\widehat{m_2}+\widehat{q}+\widehat{s}$

Fig. 5. Example of parallel cost

Example 8. Figure 5(left) shows a simple method m that spawns two tasks by calling p and q at locations x and y, resp. In turn, p spawns a task by calling s at location y. This program only features distributed execution, concurrent behaviours within the locations are ignored for now. In the sequel we denote by $\widehat{m}$ the cost of block m. $\widehat{m_1}$, $\widehat{m_2}$ and $\widehat{m_3}$ denote, resp., the cost from the beginning of m to the call x.p(), the cost between x.p() and y.q(), and the remaining cost of m. $\widehat{p_1}$ and $\widehat{p_2}$ are analogous. The resource analysis described in Sect. 2 can be used for obtaining an upper bound of the cost of each block.

The notion of parallel cost $\mathcal{P}$ corresponds to the cost consumed between the first instruction executed by the program at the initial location and the last instruction executed at any location by taking into account the parallel execution of instructions and idle times at the different locations.

Example 9. Figure 5(right) shows three possible traces of the execution of this example (more traces are feasible). Below the traces, the expressions $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ show the parallel cost for each trace. The main observation here is that the parallel cost varies depending on the duration of the tasks. It will be the worst (maximum) value of such expressions, that is, $\mathcal{P}{=}max(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \ldots)$. In ② p_1 is shorter than m_2, and s executes before q. In ③, q is scheduled before s, resulting in different parallel cost expressions. In ①, the processor of location y becomes idle after executing s and must wait for task q to arrive.

In the general case, the inference of parallel cost is complicated because: (1) It is unknown if the processor is available when we spawn a task, as this depends on the duration of the tasks that were already in the queue; e.g., when task q is spawned we do not know if the processor is idle (trace ①) or if it is taken (trace ②). Thus, all scenarios must be considered; (2) Locations can be dynamically created, and tasks can be dynamically spawned among the different locations (e.g., from location o we spawn tasks at two other locations).

Besides, tasks can be spawned in a circular way; e.g., task `s` could make a call back to location `x`; (3) Tasks can be spawned inside loops, we might even have non-terminating loops that create an unbounded number of tasks. We use a *distributed flow graph* (DFG) to capture the different flows of execution that the distributed system can perform. We use the standard partitioning of methods into blocks used to build the control flow graph of the program. The nodes in the DFG are the blocks of the CFG combined with the location's identity and are used as cost centers when obtaining the upper bound as in Sect. 2. The edges represent the control flow in the sequential execution (drawn with normal arrows) and all possible orderings of tasks in the location's queues (drawn with dashed arrows) since, when the processor is released, any pending task of the same location could start executing.

Example 10. Figure 6 shows the DFG for the program in Fig. 5. Nodes in gray are the exit nodes of the methods, and it implies that the execution can terminate executing $o{:}m_3$, $x{:}p_2$, $y{:}s$ or $y{:}q$. Solid edges include those existing in the CFG of the sequential program but combined with the location's identity and those derived from calls. The dashed edges model that the execution order of `s` and `q` at location y is unknown.

Our analysis consists of obtaining the maximal parallel cost from all possible executions of the program, based on the DFG. The execution paths in the DFG start in the initial node that corresponds to the entry method of the program, and finish in any exit node of a method. The key idea to obtain the parallel cost from paths in the graph is that the cost of each block contains not only the cost of the block itself but this cost is multiplied by the number of times the block is visited. As the order in which blocks are executed is not relevant for the resource analysis, we use sets instead of sequences. The parallel cost of the distributed system can be over-approximated by the maximum cost for all paths to nodes that correspond to method exit blocks.

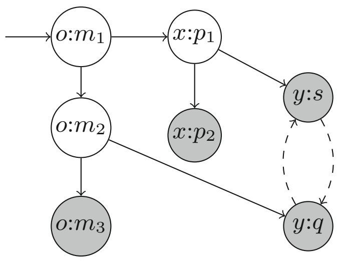


Fig. 6. DFG for Fig. 5

Example 11. Given the DFG in Example 10, we have the following sets:

$$\{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1},\ \underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2},\ \underbrace{\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\}}_{N_3},\ \underbrace{\{o{:}m_1, o{:}m_2, y{:}s, y{:}q\}}_{N_4}\}$$

Observe that these sets represent traces of the program. The execution captured by N_1 corresponds to trace ① of Fig. 5. In this trace, the code executed at location `o` leads to the maximal cost. Similarly, the set N_3 corresponds to trace ② and N_4 corresponds to trace ③. The set N_2 corresponds to a trace where $x{:}p_2$ leads to the maximal cost (not shown in Fig. 5). The cost is obtained by using the block-level costs for all nodes that compose the sets above. The overall parallel cost is computed as: $\widehat{\mathcal{P}}(\mathsf{m}(n)) = max(\mathsf{m}^+(n)|_{N_1}, \mathsf{m}^+(n)|_{N_2}, \mathsf{m}^+(n)|_{N_3}, \mathsf{m}^+(n)|_{N_4})$. Importantly, $\widehat{\mathcal{P}}$ is more precise than the serial cost because all paths have at least one missing node. For instance, N_1 does not contain the cost of $x{:}p_1$, $x{:}p_2$, $y{:}s$, $y{:}q$ and N_3 does not contain the cost of $o{:}m_2$, $o{:}m_3$, $x{:}p_2$.

6 Conclusions and Future Work

We have presented the basic concepts underlying the resource analysis of distributed systems. The overall framework is based on the idea of having cost centers which allow defining the required level of granularity. We have seen how, using cost centers, performance indicators can be defined to assess the overall performance of the distributed system, e.g., whether the load is well-balanced among the nodes, the communication costs, etc. Also, new notions of cost can be defined to estimate the peak cost required by each distributed node, and the parallel cost which exploits the parallelism in the execution.

In future work, we are investigating the new challenges that arise in the resource analysis of concurrent systems. In particular, new technique are required to infer the cost when each distributed component allows interleaving among the tasks that it has to execute. Also, we are improving the precision of the *may-happen-in-parallel* analysis which is used to infer the cost of the tasks running concurrently.

Acknowledgments. We are grateful to all members of the COSTA group and Einar Broch Johnsen for their collaborations to develop some of the analyses that underlie our basic framework and their contributions to the SACO system. 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. Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Puebla, G., Román-Díez, G.: Object-sensitive cost analysis for concurrent objects. Softw. Test. Verification Reliab. **25**(3), 218–271 (2015)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. J. Autom. Reasoning **46**(2), 161–203 (2011)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoret. Comput. Sci. Spec. Issue Quant. Aspects Program. Lang. **413**(1), 142–159 (2012)
4. Albert, E., Correas, J., Johnsen, E.B., Román-Díez, G.: Parallel cost analysis of distributed systems. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 275–292. Springer, Heidelberg (2015)
5. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Quantified abstractions of distributed systems. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 285–300. Springer, Heidelberg (2013)

6. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Quantified abstract configurations of distributed systems. Formal aspects Comput. **27**(4), 665–699 (2015)
7. 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)
8. Albert, E., Fernández, J.C., Román-Díez, G.: Non-cumulative resource analysis. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 85–100. Springer, Heidelberg (2015)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the POPL 1978, pp. 84–96 (1978)
10. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
11. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: Proceedings of POPL 2009, pp. 127–139. ACM (2009)
12. Hoffmann, J., Aehlig, K., Hofmannn, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14:1–14:62 (2012)
13. Hoffmann, J., Shao, Z.: Automatic static cost analysis for parallel programs. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 132–157. Springer, Heidelberg (2015)
14. 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)
15. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**, 1–41 (2005)
16. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the POPL 2011, pp. 17–30. ACM (2011)
17. Sutter, H., Larus, J.R.: Software and the concurrency revolution. ACM Queue **3**(7), 54–62 (2005)
18. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (1975)
19. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011)