Peak Cost Analysis of Distributed Systems Author's version**

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

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

Abstract. We present a novel static analysis to infer the *peak cost* of distributed systems. The different locations of a distributed system communicate and coordinate their actions by posting tasks among them. Thus, 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*. The peak cost of a distributed location refers to the maximum cost that it needs to carry out along its execution. Inferring the peak cost is challenging because it increases and decreases along the execution, unlike the standard notion of *total cost* which is cumulative. Our key contribution is the novel notion of *quantified queue configuration* which captures the worst-case cost of the tasks that may be simultaneously pending to execute at each location along the execution. A prototype implementation demonstrates the accuracy and feasibility of the proposed peak cost analysis.

1 Introduction

Distributed systems are increasingly used in industrial processes and products, such as manufacturing plants, aircraft and vehicles. For example, many control systems are decentralized using a distributed architecture with different processing locations interconnected through buses or networks. The software in these systems typically consists of concurrent tasks which are statically allocated to specific locations for processing, and which exchange messages with other tasks at the same or at other locations to perform a collaborative work. A decentralized approach is often superior to traditional centralized control systems in performance, capability and robustness. Systems such as control systems are often critical: they have strict requirements with respect to timing, performance, and stability. A failure to meet these requirements may have catastrophic consequences. To verify that a given system is able to provide the required quality,

** Appeared at *Proc. 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014*. The final publication is available at http://link.springer.com/chapter/10.1007%2F978-3-319-10936-7_2

an essential aspect is to accurately predict *worst-case costs*. We develop our analysis for a generic notion of cost that can be instantiated to the number of executed instructions (considered as the best abstraction of time for software), the amount of memory created, the number of tasks posted to each location, or any other *cost model* that assigns a non-negative cost to each instruction.

Existing cost analyses for distributed systems infer the *total* resource consumption [3] of each distributed location, e.g., 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 unfortunately a too pessimistic estimation of the amount of resources actually required in the real execution. An important observation is that the *peak* cost will depend on whether the tasks that the location has to execute are pending *simultaneously*. We aim at inferring such peak of the resource consumption which captures the maximum amount of resources that the location might require along any execution. In addition to its application to verification as described above, 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.

This paper presents, to the best of our knowledge, the first static analysis to infer the peak of the resource consumption of distributed systems, which takes into account the type and amount of tasks that the distributed locations can have in their queues simultaneously along any execution, to infer precise bounds on the peak cost. Our analysis works in three steps: (1) *Total cost analysis.* The analysis builds upon well-established analyses for total cost [9,3,18]. We assume that an underlying total cost analysis provides a *cost* for the tasks which measures their efficiency. (2) *Queues configurations.* The first contribution is the inference of the *abstract queue configuration* for each distributed component, 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. We rely on the information gathered by a *may-happen-in-parallel* analysis [7,1,11,5] to define the abstract queue configurations. (3) *Peak cost.* Our key contribution is the notion of *quantified queue configuration*, which over-approximates the peak cost of each distributed location. For a given queue configuration, its quantified configuration is computed by removing from the total cost inferred in (1) those tasks that do not belong to its configuration, as inferred in (2). The peak for the location is the maximum of the costs of all configurations that its queue can have.

We demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer of peak cost within the SACO system [2], a static analyzer for distributed concurrent programs. In preliminary experiments on some typical applications for distributed programs, the peak cost achieves

gains up to 70% w.r.t. a total cost analysis. The tool can be used on-line from a web interface available at http://costa.ls.fi.upm.es/web/saco.

2 The Distributed Model

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered queue of pending tasks. Initially all processors are idle. When an idle processor's task queue is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the queues of any processor, including its own, and synchronize with the completion of tasks. When a task completes or when it is awaiting for another task to terminate, its processor becomes idle again, chooses the next pending task, and so on.

2.1 Syntax

The number of distributed locations needs not be known a priori (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 program is composed by a set of methods defined as $M ::= T\ m(\bar{T}\ \bar{x})\{s\}$ where $s ::= s; s \mid x{=}e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{return} \mid b{=}\mathsf{newLoc} \mid f{=}b!m(\bar{e}) \mid \mathsf{await}\ f?$. The notation $\bar{T}$ is used as a shorthand for $T_1, \ldots, T_n$, and similarly for other names. The special location identifier *this* denotes the current location. For the sake of generality, the syntax of expressions e and types T is left open. The semantics of future variables f and concurrency instructions is explained below.

2.2 Semantics

A *program state* has the form $loc_1 \| \ldots \| loc_n$, denoting the currently existing distributed locations. Each *location* is a term $loc(lid, tid, \mathcal{Q})$ where *lid* is the location identifier, *tid* is the identifier of the *active task* which holds the location's lock or $\perp$ if the lock is free, and $\mathcal{Q}$ is the set of tasks at the location. Only one task, which holds the location's *lock*, can be *active* (running) at this location. All other tasks are *pending*, waiting to be executed, or *finished*, if they terminated and released the lock. Given a location, its set of *ready* tasks is composed by the tasks that are pending and the one that it is active at the location. A *task* is a term $tsk(tid, m, l, s, c)$ where *tid* is a unique task identifier, m is the name of the method executing in the task, l is a mapping from local variables to their values, s is the sequence of instructions to be executed or $s = \tau$ if the task has terminated, and c is a positive number which corresponds to the cost of the instructions executed in the task so far. The cost of executing an instruction i is represented in a generic way as $cost(i)$.

The execution of a program starts from a method m in an initial state S_0 with a single (initial) location of the form $S_0{=}loc(0, 0, \{tsk(0, \mathsf{m}, l, body(\mathsf{m}), 0)\})$. Here, l maps parameters to their initial values and local references to null (standard initialization), and $body(m)$ refers to the sequence of instructions in the

$$\frac{t = tsk(tid, m, l, \langle x = \mathsf{newLoc}; s\rangle, c),\ \mathrm{fresh}(lid_1)\ ,\ l' = l[x \to lid_1]}{loc(lid, tid, \{t\}\cup\mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s, c{+}cost(\mathsf{newLoc}))\}\cup\mathcal{Q}) \| loc(lid_1, \bot, \{\})} \quad (\textsc{newLoc})$$

$$\frac{l(x) = lid_1,\ \mathrm{fresh}(tid_1),\ l_1{=}buildLocals(\bar{z}, m_1),\ l' = l[f \to tid_1]}{\begin{array}{c} loc(lid, tid, \{tsk(tid, m, l, \langle f{=}x!m_1(\bar{z}); s\rangle, c)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \mathcal{Q}') \rightsquigarrow \\ loc(lid, tid, \{tsk(tid, m, l', s, c + cost(f{=}x!m_1(\bar{z})))\}\cup\mathcal{Q}) \parallel \\ loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1), 0) \cup \mathcal{Q}'\}) \end{array}} \quad (\textsc{async})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle, c), l(f) = tid_1, tsk(tid_1, \_, \_, s_1, \_) \in \mathsf{Locs}, s_1 = \tau}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, s, c + cost(\mathsf{await}\ f?))\} \cup \mathcal{Q})} \quad (\textsc{await-t})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle, c), l(f) = tid_1, tsk(tid_1, \_, \_, s_1, \_) \in \mathsf{Locs}, s_1 \neq \tau}{loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle, c)\} \cup \mathcal{Q})} \quad (\textsc{await-f})$$

$$\frac{select(\mathcal{Q}){=}tid,\ t{=}tsk(tid, \_, \_, s, c){\in}\mathcal{Q}, s{\neq}\tau}{loc(lid, \bot, \mathcal{Q}){\rightsquigarrow}loc(lid, tid, \mathcal{Q})} \quad (\textsc{select})$$

$$\frac{t = tsk(tid, m, l, \langle \mathsf{return}; \rangle, c)}{\begin{array}{c} loc(lid, tid, \{t\} \cup \mathcal{Q}) \rightsquigarrow \\ loc(lid, \bot, \{tsk(tid, m, l, \tau, c{+}cost(\mathsf{return}))\}\cup\mathcal{Q}) \end{array}} \quad (\textsc{return})$$

Fig. 1. (Summarized) Cost Semantics for Distributed Execution

method m. The execution proceeds from the initial state S_0 by selecting *non-deterministically* one of the locations and applying the semantic rules depicted in Fig. 1. The treatment of sequential instructions is standard and thus omitted. The operational semantics $\rightsquigarrow$ is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows. In NewLoc, an active task *tid* at location *lid* creates a location lid_1 which is introduced to the state with a free lock. Async spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is added to the queue of location lid_1. The case $lid{=}lid_1$ is analogous, the new task tid_1 is simply added to $\mathcal{Q}$ of *lid*. The future variable f allows synchronizing the execution of the current task with the termination of created task. The association of the future variable to the task is stored in the local variables table l'. In Await-t, the future variable we are awaiting for points to a finished task and await can be completed. The finished task t_1 is looked up at all locations in the current state (denoted by Locs). Otherwise, Await-f yields the lock so that any other task of the same location can take it. Rule Select returns a task that is not finished, and it obtains the lock of the location. Return releases the lock and it will never be taken again by that task. Consequently, that task is *finished* (marked by adding τ). For brevity, we omit the return instructions in the examples.

3 Peak Cost of Distributed Systems

The aim of this paper is to infer an *upper bound* on the *peak cost* for all locations of a distributed system. The peak cost refers to the maximum amount

of resources that a given location might require along any execution. The over-approximation consists in computing the sum of the costs of all tasks that can be simultaneously ready in the location's queue. Importantly, as the number of ready tasks in the queue can increase and decrease along the execution, in order to define the notion of peak cost, we need to observe all intermediate states along the computation and take the maximum of their costs.

Example 1. Figure 2 shows to the left a method m that spawns several tasks at a location referenced from variable x (the middle code can be ignored by now). To the right of the figure, we depict the tasks that are ready in the queue of location x at different states of the execution of m. For instance, the state ① is obtained after invoking method r at line 2 (L2 for short). The first iteration of the while loop spawns a task p (state ②) and then invokes q (state ③). The important observation is that q is awaited at L6, and thus it is guaranteed to be finished at state ④. The same pattern is repeated in subsequent loop iterations (states ⑤ to ⑦). The last iteration of the loop, captured in state ⑦, accumulates all calls to p, and the last call to q. Observe that at most one instance of method q appears in the queue at any point during the execution of the program. Finally, state ⑧ represents the exit of the loop (L8) and ⑨ when method s is invoked at L9. The await at L6 ensures that methods q and s will not be queued simultaneously.

We start by formalizing the notion of peak cost in the concrete setting. Let us provide some notation. Given a state $S{\equiv}loc_1 \| \ldots \| loc_n$, we use $loc \in S$ to refer to a location in S. The set of ready tasks at a location lid at state S is defined as $ready\_tasks(S, lid) = \{tid \mid loc(lid, \_, \mathcal{Q}) \in S, tsk(tid, \_, \_, s, \_){\in}\mathcal{Q}, s{\neq}\tau\}$. Note that we exclude the tasks that are finished. Given a finite trace $t \equiv S_0{\rightarrow}\ldots{\rightarrow}S_N$, we use $\mathcal{C}(lid, tid)$ to refer to the accumulated cost c in the final state S_N by the task $tsk(tid, \_, \_, \_, c){\in}\mathcal{Q}$ that executes at location $loc(lid, \_, \mathcal{Q}){\in}S_N$, and $\mathcal{C}(S_i, lid)$ to refer to the accumulated cost of all active tasks that are in the queue at state S_i for location lid: $\mathcal{C}(S_i, lid) = \sum_{tid \in ready\_tasks(S_i, lid)} \mathcal{C}(lid, tid)$. Now, the peak cost of location lid is defined as the *maximum* of the addition of the costs of the tasks that are simultaneously ready at the location at any state: $peak\_cost(t, lid) = \max(\{\mathcal{C}(S_i, lid) \mid S_i \in t\})$. Observe that the *cost* always refers to the cost of each task in the *final* state S_N. This way we are able to capture the cost that a location will need to carry out at each state S_i with $i \leq N$ in which we have a set of ready tasks in its queue but they have (possibly) not been executed yet.

Since execution is non-deterministic in the selection of tasks, given a program $P(x)$, multiple (possibly infinite) traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible traces for $P(\overline{x})$.

Definition 1 (peak cost). *The peak cost of a location with identifier lid in a program P on input values $\overline{x}$, denoted $\mathcal{P}(P(\overline{x}), lid)$, is defined as:*

$\mathcal{P}(P(\overline{x}), lid) = max(\{peak\_cost(t, lid) \mid t \in executions(P(\overline{x}))\})$

Example 2. Let us reason on the peak cost for the execution of m. We use $\ddot{m}$ to refer to the concrete cost of a task executing method m. We use subscripts $\ddot{m}_j$ to refer to the cost of the j-th task spawned executing method m. As the cost

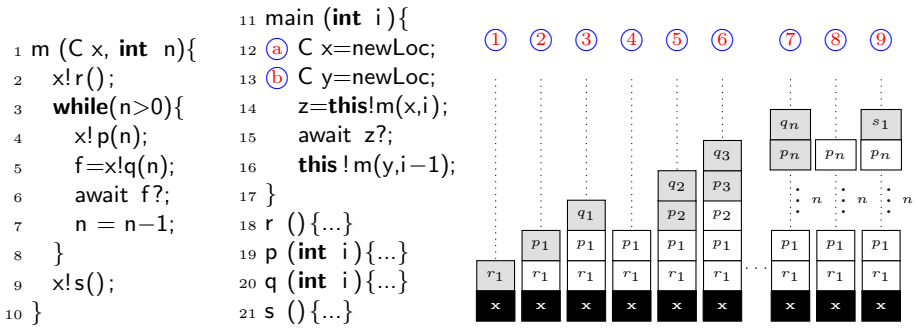

Fig. 2. Running example

often depends on the parameters, in general, we have different costs $\dddot{m}_1$, $\dddot{m}_2$,... for the multiple executions of the same method. The queue of x in states ② and ④ accumulates the cost $\dddot{r}_1 + \dddot{p}_1$. At ⑥, it accumulates $\dddot{r}_1 + \dddot{p}_1 + \dddot{p}_2 + \dddot{p}_3 + \dddot{q}_3$. The peak cost corresponds to the maximum among all states. Note that it is unknown if the cost at ⑦ is larger than the cost at ③-⑤-⑥-.... This is because at each state we have a different instance of q running, and it can be that $\dddot{q}_1$ is larger than the whole cost of the next iterations. Only some states can be discarded (for instance ① and ② are subsumed by ③, and ⑧ by ⑨).

The above example reveals several important aspects that make the inference of the peak cost challenging: (1) We need to infer all possible queue configurations. This is because the peak cost is non-cumulative, and any state can require the maximum amount of resources and constitute the peak cost. This contrasts with the total cost in which we only need to observe the final state. (2) We need to track when tasks terminate their execution and eliminate them from the configuration (the await instructions will reveal this information). (3) We need to know how many instances of tasks we might have running and bound the cost of each instance, as they might not all have the same cost.

4 Basic Concepts: Points-to, Cost, and MHP Analyses

Our peak cost analysis builds upon well-established work on points-to analysis [14,13], total cost analysis [9,18,3] and may-happen-in-parallel (MHP) analysis [11,5]. As regards the points-to and may-happen-in-parallel analyses, this section only reviews the basic concepts which will be used later by our peak cost analysis. As for the total cost analysis, we need to tune several components of the analysis in order to produce the information that the peak cost analysis requires.

Points-to Analysis. Since locations can be dynamically created, we need an analysis that abstracts them into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. Points-to analysis [14,13,16] solves this problem. It infers the set of memory locations that a reference variable can *point-to*. Different abstractions can be used and our method

is parametric on the chosen abstraction. Any points-to analysis that provides the following information with more or less accurate precision can be used (our implementation uses [13]): (1) $\mathcal{O}$, the set of abstract locations; (2) $\mathcal{M}(o)$, the set of methods executed in tasks at the abstract location $o \in \mathcal{O}$; (3) a function $pt(pp, v)$ which for a given program point pp and a variable v returns the set of abstract locations in $\mathcal{O}$ to which v may point to.

Example 3. Consider the main method shown in Fig. 2, which creates two new locations x at program point ⓐ (abstracted as o_1) and y at ⓑ (abstracted as o_2) and passes them as parameters in the calls to m (at L14, L16). By using the points-to analysis we obtain the following relevant information, $\mathcal{O}$=$\{\epsilon, o_1, o_2\}$ where ϵ is the location executing main, $\mathcal{M}(o_1)$=$\{r, p, q, s\}$, $\mathcal{M}(o_2)$=$\{r, p, q, s\}$, $pt(L14, x)$=$\{o_1\}$ and $pt(L16, y)$=$\{o_2\}$. Observe that the abstract task executing p at location o_1 represents multiple instances of the tasks invoked at L4.

Cost Analysis. The notion of *cost center* (CC) is an artifact used to define the granularity of a cost analyzer. In [3], the proposal is to define a CC for each distributed location, i.e., CCs are of the form $c(o)$ where $o \in \mathcal{O}$. In essence, the analyzer every time that accounts for the cost of executing an instruction b at program point pp, it also checks at which locations it is executing. This information is provided by the points-to analysis as O_{pp}=$pt(pp, this)$. The cost of the instruction is accumulated in the CCs of all elements in O_{pp} as $\sum c(o)*cost(b), \forall o \in O_{pp}$, where $cost(b)$ expresses in an abstract way the cost of executing the instruction. If we are counting steps, then $cost(b) = 1$. If we measure memory, $cost(b)$ refers to memory created by b. Then, given a method $m(\bar{x})$, the cost analyzer computes an *upper bound* for the total cost of executing m of the form $\widehat{\mathcal{C}}_m(\bar{x})$=$\sum_{i=1}^{n} c(o_i)*C_i$, where o_i∈$\mathcal{O}$ and C_i is a cost expression that bounds the cost of the computation carried out by location o_i when executing m. We omit the subscript in $\widehat{\mathcal{C}}$ when it is clear from the context. Thus, CCs allow computing costs at the granularity level of the distributed locations. If one is interested in studying the computation performed by one particular location o_j, denoted $\widehat{\mathcal{C}}_m(\bar{x})|_{o_j}$, we simply replace all $c(o_i)$ with i≠j by 0 and $c(o_j)$ by 1. The use of CCs is of general applicability and different approaches to cost analysis (e.g., cost analysis based on recurrence equations [17], invariants [9] or type systems [10]) can trivially adopt this idea so as to extend their frameworks to a distributed setting. In principle, our method can work in combination with any analysis for total cost (except for the accuracy improvement in Sec. 5.3).

Example 4. By using the points-to information obtained in Ex. 3, a cost analyzer (we use in particular [2]) would obtain the following upper bounds on the cost distributed at the locations o_1 and o_2 (we ignore location ϵ in what follows as it is not relevant): $\widehat{\mathcal{C}}_{main}(i)$=$c(o_1)*\widehat{r_1} + c(o_1)*i*\widehat{p_1} + c(o_1)*i*\widehat{q_1} + c(o_1)*\widehat{s_1} + c(o_2)*\widehat{r_2} + c(o_2)*(i-1)*\widehat{p_2} + c(o_2)*(i-1)*\widehat{q_2} + c(o_2)*\widehat{s_2}$. There are two important observations: (1) the analyzer computes the *worst-case* cost $\widehat{p_1}$ for all instances of tasks spawned at L4 executing p at location o_1 (note that it is multiplied by the number of iterations of the loop "i"); (2) the upper bound at location o_2 for

the tasks executing p is $\widehat{p_2}$, and it is different from $\widehat{p_1}$ as the invocation to m at L16 has different initial parameters. By replacing $c(o_1)$ by 1 we obtain the cost executed at the location identified by o_1, that is, $\widehat{\mathcal{C}}_{main}|_{o_1}=\widehat{r_1}+i*\widehat{p_1}+i*\widehat{q_1}+\widehat{s_1}$.

Context-Sensitive Task-level Cost Centers. Our only modification to the total cost analysis consists in using *context-sensitive task-level* granularity by means of appropriate CCs. Let us first focus on the task-level aspect. We want to distinguish the cost of the tasks executing at the different locations. We define task-level cost centers, $\overline{\mathcal{T}}$, as the set $\{o{:}m \in \mathcal{O} \times \mathcal{M} \mid o \in pt(pp, this) \wedge pp \in m\}$, which contains all methods combined with all location names that can execute them. In the example, $\overline{\mathcal{T}}=\{\epsilon{:}m, o_1{:}r, o_1{:}p, o_1{:}q,\ o_1{:}s, o_2{:}r, o_2{:}p, o_2{:}q, o_2{:}s\}$. Now, the analyzer every time that accounts for the cost of executing an instruction *inst*, it checks at which location it is executing (e.g., o) and to which method it belongs (e.g., m), and it accumulates $c(o{:}m)*cost(b)$. Thus, it is straightforward to modify an existing cost analyzer to include task-level cost centers. The context-sensitive aspect refers to the fact that the whole cost analysis can be made context-sensitive by considering the calling context when analyzing the tasks [15]. As usual, the context is the *chain of call sites* (i.e., the program point in which the task is spawned and those of its ancestor calling methods). The length of the chains is up to a maximum k which is a fixed parameter of the analysis. For instance, for k=2, we distinguish $\mathsf{14{:}4{:}p}$ the task executing p from the first invocation to m at L14 and $\mathsf{16{:}4{:}p}$ the one arising from L16. Their associated CCs are then $o_1{:}\mathsf{14{:}4{:}p}$ and $o_2{:}\mathsf{16{:}4{:}p}$. In the formalization, we assume that the context (call site chain) is part of the method name m and thus we write CCs simply as $c(o{:}m)$. Then, given an entry method $p(\bar{x})$, the cost analyzer will compute a *context-sensitive task-level upper bound* for the cost of executing p of the form $\widehat{\mathcal{C}}_p(\bar{x})=\sum_{i=1}^{n} c(o_i{:}m_i)*C_i$, where $o_i{:}m_i \in \overline{\mathcal{T}}$, and C_i is a cost expression that bounds the cost of the computation carried out by location o_i executing method m_i, where m_i contains the calling context. The notation $\widehat{\mathcal{C}}_p(\bar{x})|_{o:m}$ is used to obtain the cost associated with $c(o{:}m)$ within $\widehat{\mathcal{C}}_p(\bar{x})$, i.e., the one obtained by setting to zero all $c(o'{:}m')$ with $o'{\neq}o$ or $m'{\neq}m$ and to one $c(o{:}m)$.

Example 5. For the method main shown in Fig. 2, the cost expression obtained by using task-level CCs and k=0 (i.e., making it context insensitive) is the following: $\widehat{\mathcal{C}}(i)=c(o_1{:}r)*\widehat{r_1}+c(o_1{:}p)*i*\widehat{p_1}+c(o_1{:}q)*i*\widehat{q_1}+c(o_1{:}s)*\widehat{s_1}+c(o_2{:}r)*\widehat{r_2}+c(o_2{:}p)*(i{-}1)*\widehat{p_2}+c(o_2{:}q)*(i{-}1)*\widehat{q_2}+c(o_2{:}s)*\widehat{s_2}$. To obtain the cost carried out by o_1 when executing q, we replace $c(o_1{:}q)$ by 1 and the remaining CCs by 0, resulting in $\widehat{\mathcal{C}}(i)|_{o_1:q}=i*\widehat{q_1}$. For k>0, we simply add the call site sequences in the CCs, e.g., $c(o_1{:}\mathsf{14{:}4{:}p})$.

May-Happen-in-Parallel Analysis. We use a MHP analysis [11,5] as a black box and assume the same context and object-sensitivity as in the cost analysis. We require that it provides us: (1) The set of MHP pairs, denoted $\tilde{\mathcal{E}}_P$, of the form $(o_1{:}p_1, o_2{:}p_2)$ which indicates that program point p_1 running at location o_1 and program point p_2 running at location o_2 might execute in parallel. (2) A function $nact(o{:}m)$ that returns 1 if only one instance of m can be active at location o or ∞ if we might have more than one ([5] provides both 1 and 2).

Example 6. An MHP analysis [5] infers for the main method in Fig. 2, among others, the following set of MHP pairs at location o_1, $\{(o_1{:}18, o_1{:}19), (o_1{:}18, o_1{:}20), (o_1{:}18, o_1{:}21), (o_1{:}19, o_1{:}20), (o_1{:}19, o_1{:}21)\}$. In essence, each pair is capturing that the corresponding methods might happen in parallel, e.g., $(o_1{:}18, o_1{:}19)$ implies that methods r and p might happen in parallel. The MHP analysis learns information from the await to capture that only one instance of q can be active at location o_1, thus $nact(o_1{:}q){=}1$. On the contrary, the number of active calls to p is greater than 1, then $nact(o_1{:}p){=}\infty$.

5 Peak Cost Analysis

In this section we present our framework to infer the peak cost. It consists of two main steps: we first infer in Sec. 5.1 the configurations that the (abstract) location queue can feature (we use the MHP information in this step); and in a second step, we compute in Sec. 5.2 the cost associated with each possible queue configuration (we use the total cost in this step). Finally, we discuss in Sec. 5.3 an important extension of the basic framework that can increase its accuracy.

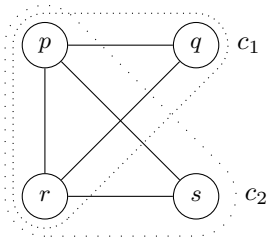

Fig. 3. $\mathcal{G}_t(o_1)$ for Fig 2

5.1 Inference of Queue Configurations

Our goal now is to infer, for each abstract location in the program, all its *non-quantified* configurations, i.e., the sets of method names that can be executing in tasks that are simultaneously ready in the location's queue at some state in the execution. Configurations are non-quantified because we ignore how many instances of a method can be pending in the queue and their costs.

Definition 2 (tasks queue graph). *Given a program P, an abstract location $o \in \mathcal{O}$ and the results of the MHP analysis $\tilde{\mathcal{E}}_P$, the* tasks queue graph *for o $\mathcal{G}_t(o){=}\langle V_t, E_t\rangle$ is an undirected graph where $V_t = \mathcal{M}(o)$ and $E_t = \{(m_1, m_2) \mid (p_1, p_2) \in \tilde{\mathcal{E}}_P, p_1{\in}m_1, p_2{\in}m_2, m_1{\neq}m_2\}$.*

It can be observed in the above definition that when we have two program points that may-happen-in-parallel in the location's queue, then we add an edge between the methods to which those points belong.

Example 7. By using the MHP information for location o_1 in Ex. 6, we obtain the tasks queue graph $\mathcal{G}_t(o_1)$ shown in Fig. 3 with the following set of edges $\{(r,p), (r,q), (r,s), (p,s), (p,q)\}$ (dotted lines will be explained later).

The tasks queue graph allows us to know the sets of methods that may be ready in the queue simultaneously. This is because, if two methods might be queued at the same time, there must be an edge between them in the tasks queue graph. It

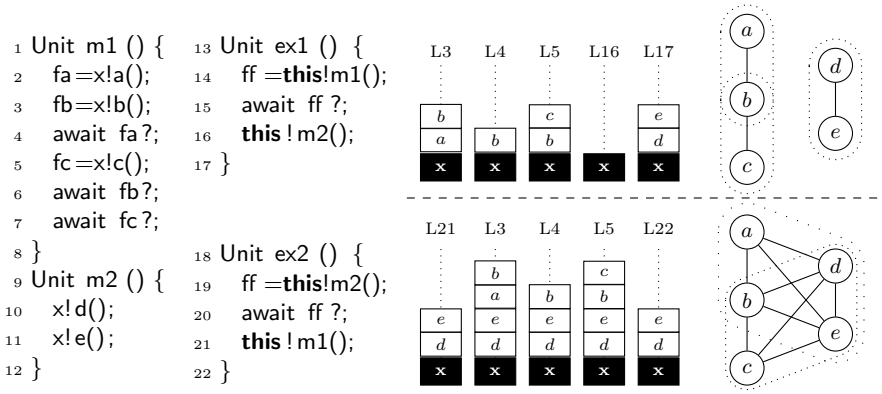

Fig. 4. Queue Configurations Example

is then possible to detect the subsets of methods that can be queued at the same: those that are connected with edges between every two nodes that represent such subset, i.e., they form a *clique*. Since we aim at finding the maximum number of tasks that can be queued simultaneously, we need to compute the *maximal cliques* in the graph. Formally, given an undirected graph $\mathcal{G}{=}\langle V, E\rangle$, a *maximal clique* is a set of nodes $C{\subseteq}V$ such that every two nodes in C are connected by an edge, and there is no other node in $V{\setminus}C$ connected to every node in C.

Example 8. For $\mathcal{G}_t(o_1)$ in Fig. 3, we have two maximal cliques: $c_1 = \{p, q, r\}$ and $c_2 = \{p, r, s\}$, which capture the states ⑦ and ⑨ of the queue of o_1 (see Fig. 2). Observe that the maximal cliques subsume other states that contain subsets of a maximal clique. For instance, states ①-⑥ are subsumed by c_1.

Definition 3 (queue configuration). *Given a location o, we define its queue configuration, denoted by $\mathcal{K}(o)$, as the set of maximal cliques in $\mathcal{G}_t(o)$.*

Therefore, a queue configuration is a set of sets, namely each element in $\mathcal{K}(o)$ is a set of method names which capture a possible configuration of the queue. Clearly, all possible (maximal) configurations must be considered in order to obtain an over-approximation of the peak cost.

Example 9. Let us see a more sophisticated example for queue configurations. Consider the methods in Fig. 4 which have two distinct entry methods, ex1 and ex2. They both invoke method m1, which spawns tasks a, b and c. m1 guarantees that a, b and c are completed when it finishes. Besides, we know that b and c might run in parallel, while the await instruction in L4 ensures that a and c cannot happen in parallel. Method m2 spawns tasks d and e and does not await for their termination. We show in the middle of Fig. 4 the different configurations of the queue of x (at the program points marked on top) when we execute ex1 (above) and ex2 (below). Such configurations provide a graphical view of the results of the MHP analysis (which basically contains pairs for each

two elements in the different queue states). In the queue of ex1, we can observe that the await instructions at the end of m1 guarantee that the queue is empty before launching m2 (see queue at L16). To the right of the queue we show the resulting tasks queue graph for ex1 obtained by using the MHP pairs which correspond to the queues showed in the figure. Then, we have $\mathcal{K}(x)$=$\{\{a,b\}, \{b,c\},\{d,e\}\}$. Note that these cliques capture the states of the queue at L3, L5 and L17, respectively. As regards ex2, the difference is that m2 is spawned before m1. Despite the await at L21, m2 is not awaiting for the termination of d nor e, thus at L21 the queue might contain d and e. As for m1, we have a similar behaviour than before, but now we have to accumulate also d and e along the execution of m1. The resulting tasks queue graph is showed to the right. It can be observed that it is densely connected, and now $\mathcal{K}(x)$=$\{\{d,e,a,b\},\{d,e,b,c\}\}$. Such cliques correspond to the states of the queue at L3 and L5, respectively.

5.2 Inference of Quantified Queue Configurations

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, (2) the worst-case cost of such instances. The main observation is that the upper bounds on the total cost in Sec. 4 already contain both types of information. This is because the cost attached to the CC $c(o{:}m)$ accounts for the accumulation of the resource consumption of *all* tasks running method m at location o. We therefore can safely use $\widehat{\mathcal{C}}(\overline{x})|_{o:m}$ as upper bound of the cost associated with the execution of method m at location o.

Example 10. According to Ex. 5, the costs accumulated in the CCs of $o_1{:}q$ and $o_1{:}p$ are $\widehat{\mathcal{C}}(i)|_{o_1:p} = i * \widehat{p}$ and $\widehat{\mathcal{C}}(i)|_{o_1:q} = i * \widehat{q}$. Note that $o_1{:}q$ is accumulating the cost of *all* calls to q, as the fact that there is at most one active call to q is not taken into account by the total cost analysis. This is certainly a sound but imprecise over-approximation that will be improved in Sec. 5.3.

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 $\mathcal{K}(o)$ and stay with the maximum of all of them. Given an abstract location o and a clique $k \in \mathcal{K}(o)$, we have that $\widehat{\mathcal{C}}(\overline{x})|_k = \sum_{m\in k} \widehat{\mathcal{C}}(\overline{x})|_{o:m}$ is the cost for the tasks in k.

Definition 4. *Given a program $P(\overline{x})$ and an abstract location o, the peak cost for o, denoted $\widehat{\mathcal{P}}(P(\overline{x}),o)$, is defined as $\widehat{\mathcal{P}}(P(\overline{x}),o) = \max(\{\widehat{\mathcal{C}}(\overline{x})|_k \mid k \in \mathcal{K}(o)\})$.*

Intuitively, as the elements of $\mathcal{K}$ capture all possible configurations that the queue can take, it is sound to take the maximum cost among them.

Example 11. Following Ex. 8, the quantified queue configuration, that gives the peak cost, accumulates the cost of all nodes in the two cliques, $\widehat{\mathcal{C}}(i)|_{c_1}$=$\widehat{r}$+$i$*$\widehat{p}$+$i$*$\widehat{q}$ and $\widehat{\mathcal{C}}(i)|_{c_2}$=$\widehat{r}$+$i$*$\widehat{p}$+$\widehat{s}$. The maximum between both expressions captures the peak cost for o_1, $\widehat{\mathcal{P}}(main(i),o_1) = max(\ \{\widehat{r}{+}i{*}\widehat{p}{+}i{*}\widehat{q}, \widehat{r}{+}i{*}\widehat{p}{+}\widehat{s}\})$.

The following theorem states the soundness of our approach.

Theorem 1 (soundness). *Given a program P with arguments $\overline{x}$, a concrete location lid, and its abstraction o, we have that $\mathcal{P}(P(\overline{x}), lid) \leq \widehat{\mathcal{P}}(P(\overline{x}), o)$.*

5.3 Number of Tasks Instances

As mentioned above, the basic approach has a weakness. From the queue configuration, we might know that there is at most one task running method m at location o. However, if we use $\widehat{\mathcal{C}}(\overline{x})|_{o:m}$, we are accounting for the cost of all tasks running method m at o. We can improve the accuracy as follows. First, we use an instantiation of the cost analysis in Sec. 4 to determine how many instances of tasks running m at o we might have. This can be done by defining function *cost* in Sec. 4 as follows: $cost(inst) = 1$ if $inst$ is the entry instruction to a method, and 0 otherwise. We denote by $\widehat{\mathcal{C}}^c(\overline{x})$ the upper bound obtained using such cost model that counts the number of tasks spawned along the execution, and $\widehat{\mathcal{C}}^c(\overline{x})|_{o:m}$ the number of tasks executing m at location o.

Example 12. The expression that bounds the number of calls from main is $\widehat{\mathcal{C}}^c(i) = c(o_1{:}r) + i{*}c(o_1{:}p) + i{*}c(o_1{:}q) + c(o_1{:}s) + c(o_2{:}r) + (i-1){*}c(o_2{:}p) + (i-1){*}c(o_2{:}q) + c(o_2{:}s)$. It can be seen that CCs are the same as the ones used in Ex. 5. The difference is that when inferring the number of calls we do not account for the cost of each method but rather count 1. Then, $\widehat{\mathcal{C}}^c(i)|_{o_1:q} = i$ and $\widehat{\mathcal{C}}^c(i)|_{o_2:q} = i-1$.

Let us assume that the same cost analyzer has been used to approximate $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{C}}^c$, and that the analysis assumed the worst-case cost of m for *all* instances of m. Then, we can gain precision by obtaining the cost as $\widetilde{\mathcal{C}}(\overline{x})|_{o:m} = \widehat{\mathcal{C}}(\overline{x})|_{o:m} / \widehat{\mathcal{C}}^c(\overline{x})|_{o:m}$ if $nact(o{:}m) = 1$ and $\widetilde{\mathcal{C}}(\overline{x})|_{o:m} = \widehat{\mathcal{C}}(\overline{x})|_{o:m}$, otherwise. Intuitively, when the MHP analysis tells us that there is at most one instance of m (by means of *nact*) and, under the above assumptions, the division is achieving the desired effect of leaving the cost of one instance only.

Example 13. As we have seen in Ex. 6, the MHP analysis infers $nact(o_1{:}p) = \infty$ and $nact(o_1{:}q) = 1$. Thus, by the definition of $\widetilde{\mathcal{C}}$, the cost for p is $\widetilde{\mathcal{C}}(i)|_{o_1:p} = i * \widehat{p}$ (the same obtained in Ex. 10). However, for q we can divide the cost accumulated by all invocations to q by the number of calls to q, $\widetilde{\mathcal{C}}(i)|_{o_1:q} = i * \widehat{q}/i = \widehat{q}$.

Unfortunately, it is not always sound to make such division. The problem is that the cost accumulated in a CC for a method m might correspond to the cost of executions of m from different calling contexts that do not necessarily have the same worst-case cost. If we divide the expression $\widehat{\mathcal{C}}(\overline{x})|_{o:m}$ by the number of instances, we are taking the average of the costs, and this is not a sound upper bound of the peak cost, as the following example illustrates.

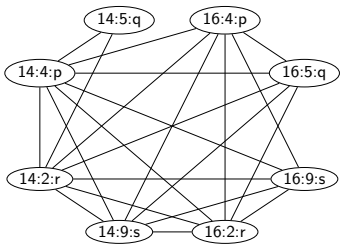

Fig. 5. Queue Config. for Fig. 2

Example 14. Consider a method main' which is as main of Fig. 2 except that we replace L16 by $\mathsf{this!m(x, i-1)}$, i.e., while main uses two different locations, x and y, in main' we only use x. Such modification affects the precision because it merges o_1 and o_2 in a single queue, o_1. Now, in main', s, launched by the first call to m, might run in parallel with q, spawned in the second call to m. Therefore, in Fig. 3 a new edge that connects q and s appears, and consequently, the new queue configuration contains all methods in just one clique $\{p, q, r, s\}$. Moreover, CCs $o_1{:}q$ with $o_2{:}q$ are merged in a single CC $o_1{:}q$. For main', the cost of $\widehat{q}$ is $\widehat{\mathcal{C}}(i)|_{o:q}{=}i{*}\widehat{q}+(i{-}1){*}\widehat{q}$, and the number of calls is $\widehat{\mathcal{C}}^c(i)|_{o:q}{=}i+(i{-}1)$. Assume that the cost of q is $\widehat{q}=n*5$ which is a function on the parameter n. The worst-case cost for $\widehat{q}$ depends on the calling context: in the context at L14, we have $\widehat{q}=i{*}5$ while in L16, we have $\widehat{q}=(i{-}1){*}5$. Then, the cost that we obtain for main' is $\widehat{\mathcal{C}}(i)|_{o:q}{=}i{*}i{*}5+(i{-}1){*}((i{-}1){*}5)$. The division of $\widehat{\mathcal{C}}(i)$ by $\widehat{\mathcal{C}}^c(i)$ is not sound because it computes the average cost of all calls to q, rather than the peak.

Importantly, we can determine when the above division is sound in a static way. The information we are seeking is within the *call graph* for the program: (1) If there are not convergence nodes in the call graph (i.e., the call graph is a tree), then it is guaranteed that we do not have invocations to the same method from different contexts. In this case, if there are multiple invocations, it is because we are invoking m from the same context within a loop. Typically, automated cost analyzers assume the same worst-case cost for all loop iterations and, in such case, it is sound to make the division. Note that if the total cost analysis infers a different cost for each loop iteration, the accuracy improvement in this section cannot be applied; (2) If there are convergence nodes, then we need to ensure that the context-sensitive analysis distinguishes the calls that arise from different points, i.e., we have different CCs for them. This can be ensured if the length of the chains of call sites used in the context by the analysis, denoted k, is larger than k_d, the depth of the subgraph of the call graph whose root is the first convergence node encountered. Note that, in the presence of recursive methods, we will not be able to apply this accuracy improvement since the depth is unbounded. Theorem 1 holds for $\widetilde{\mathcal{C}}$ if the context considered by the analysis is greater than k_d.

Theorem 2. *Let $\widetilde{\mathcal{P}}(P(\overline{x}), o)$ be the peak cost computed using $\widetilde{\mathcal{C}}$. We have that $\mathcal{P}(P(\overline{x}), lid){\leq}\widetilde{\mathcal{P}}(P(\overline{x}), o)$ if $k{>}k_d$, where k is the length of the context used.*

Example 15. Let us continue with main' of Ex. 14. Assuming that p, q, r and s do not make any further call, the call graph has m as convergence node, and thus $k_d{=}1$. Therefore, we apply the context-sensitive analysis with $k{=}2$. The context-sensitive analysis distinguishes 14:4:p, 16:4:p, and, in q, 14:5:q and 16:5:q. The queue configuration is showed in Fig. 5. In contrast to Ex. 14 we have three different cliques, $\mathcal{K}(o_1){=}\{\{$14:4:p, 14:2:r, 14:5:q$\}$, $\{$14:4:p, 14:2:r, 14:9:s, 16:4:p, 16:2:r, 16:5:q$\}$, $\{$14:4:p, 14:2:r, 14:9:s, 16:4:p, 16:2:r, 16:9:s$\}\}$, which capture more precisely the queue states (e.g., we know that 16:5:q cannot be in the queue with 16:9:s but it might be with 14:9:s). Besides, we have two different CCs for q, 14:5:q and 16:5:q, which allow us to safely apply the division

				Context Insensitive					Context Sensitive				
Bench.	loc	$\#_{\mathsf{c}}$	T	$\#_{\mathsf{q}}$	$\%_{\overline{q}}$	$\%_m$	$\%_M$	$\%_{\widehat{\mathcal{P}}}$	$\#'_{\mathsf{q}}$	$\%'_{\overline{q}}$	$\%'_m$	$\%'_M$	$\%_{\widetilde{\mathcal{P}}}$
BBuffer	107	6	2.0	9	66.7%	50.0%	100%	78.1%	10	52.2%	17.6%	100%	31.6%
MailS	97	6	2.8	8	75.1%	71.6%	100%	81.7%	8	73.3%	71.6%	100%	81.7%
DistHT	150	4	2.5	8	69.4%	53.7%	100%	88.0%	8	69.4%	46.4%	100%	88.0%
Chat	328	10	2.4	16	66.0%	50.0%	100%	90.8%	16	66.0%	7.5%	100%	90.8%
P2P	259	9	28.0	26	52.9%	91.1%	100%	97.3%	32	32.3%	44.6%	100%	64.7%
Mean					66.0%	62.3%	100%	87.1%		58.6%	37.46%	100%	71.3%

Table 1. Experimental results (times in seconds)

as to obtain the cost of a single instance of q for the two different contexts. We obtain $\widehat{\mathcal{C}}(i)|_{14:5:\mathsf{q}}{=}i{*}i{*}5$ and $\widehat{\mathcal{C}}(i)|_{16:5:\mathsf{q}}{=}(i{-}1){*}((i{-}1){*}5)$, and for the number of calls, $\widehat{\mathcal{C}}^c(i)|_{14:5:\mathsf{q}}{=}i$ and $\widehat{\mathcal{C}}^c(i)|_{16:5:\mathsf{q}}{=}i{-}1$. Using such expressions we compute $\widetilde{\mathcal{C}}(i)|_{14:5:\mathsf{q}} = i{*}5$ and $\widetilde{\mathcal{C}}(i)|_{16:5:\mathsf{q}} = (i{-}1){*}5$ which are sound and precise over-approximations for the cost due to calls to q.

6 Experimental evaluation

We have implemented our analysis in SACO [2] and applied it to some typical examples of distributed systems: BBuffer, a bounded-buffer for communicating producers and consumers; MailS, a client-server distributed system; Chat, a chat application; DistHT, a distributed hash table; and P2P, a peer-to-peer network. Experiments have been performed on an Intel Core i5 (1.8GHz, 4GB RAM), running OSX 10.8. Table 1 summarizes the results obtained. Columns Bench. and loc show, resp., the name and the number of program lines. Column $\#_{\mathsf{c}}$ shows the number of locations identified by the analysis. Columns T and $\#_{\mathsf{q}}$ show, resp., the time to perform the analysis and the number of cliques.

We aim at comparing the gain of using peak cost analysis w.r.t. total cost. Such gain is obtained by evaluating the expression that divides the peak cost by the total cost for 15 different values of the input parameters, and computing the average. The gain is computed at the level of locations, by comparing the peak cost for the location with the total cost for such location in all columns except in $\%_{\overline{q}}$, where we show the average gain at the level of cliques. Columns $\%_m$ and $\%_M$ show, resp., the greatest and smallest gain among all locations. Column $\%_{\widehat{\mathcal{P}}}$ shows the average gain weighted by the cost associated with each location (locations with higher resource consumption have greater weight). Columns $\#'_{\mathsf{q}}$, $\%'_{\overline{q}}$, $\%'_m$, $\%'_M$, and $\%_{\widetilde{\mathcal{P}}}$ contain the same information for the context-sensitive analysis. As we do not have yet an implementation of the context-sensitive analysis, we have replicated those methods that are called from different contexts. DistHT and Chat do not need replication. The last row shows the arithmetic mean of all results.

We can observe in the table that the precision gained by considering all possible queue states ($\%_{\overline{q}}$ and $\%'_{\overline{q}}$) is significant. In the context-insensitive analysis, it ranges from a gain of 53% to 75% (on average 66%). Such value is improved in the

context sensitive analysis, resulting in an average gain of 58.6%. This indicates that the cliques capture accurately the cost accumulated in the different states of the locations' queues. The gain of the context sensitive analysis is justified by the larger number of cliques ($\#'_{\mathsf{q}}$) in BBuffer and P2P. The maximal gains showed in columns $\%_m$ (and $\%'_m$) indicate that the accuracy can be improved on average 62.3% (and 37.46%). The minimal gains in $\%_M$ and $\%'_M$ are always 100%, i.e., no gain. This means that in all benchmarks we have at least one state that accumulates the cost of all methods executed at its location (typically because await is never used). Columns $\%_{\widehat{\mathcal{P}}}$ and $\%_{\widetilde{\mathcal{P}}}$ show, in BBuffer and P2P, that $\widetilde{\mathcal{P}}$ significantly outperforms $\widehat{\mathcal{P}}$. Such improvement is achieved by a more precise configuration graph that contains more cliques, and by the division on the number of calls in methods that require a significant part of the resource consumption. However, in MailS, $\widetilde{\mathcal{P}}$ does not improve the precision of $\widehat{\mathcal{P}}$. This is because the methods that contain one active instance are not part of the cliques that lead to the peak cost of the location. Despite of the NP-completeness of the clique problem, the time spent performing the clique computation is irrelevant in comparison with the time taken by the upper bound computation (less than 50ms for all benchmarks). All in all, we argue that our experiments demonstrate the accuracy of the peak cost analysis, even in its context insensitive version, with respect to the total cost analysis.

7 Conclusions, Related and Future Work

To the best of our knowledge, our work constitutes the first analysis framework for peak cost of distributed systems. This is an essential problem in the context of distributed systems. It is of great help to dimension the distributed system in terms of processing requirements, and queue sizes. Besides, it paves the way to the accurate prediction of response times of distributed locations. The task-level analysis in [4] is developed for a specific cost model that infers the peak of tasks that a location can have. There are several important differences with our work: (1) we are generic in the notion of cost and our framework can be instantiated to measure different types of cost, among them the task-level; (2) the distributed model that we consider is more expressive as it allows concurrent behaviours within each location (by means of instruction await), while [4] assumes a simpler asynchronous language in which tasks are run to completion; (3) the analysis requires the generation of non-standard recurrence equations, while our analysis benefits from the upper bounds obtained using standard recurrence equations for total cost, without requiring any modification. Indeed, the analysis in [4] could be reformulated in our framework using the MHP analysis of [11,12].

Also, the peak heap consumption in the presence of garbage collection is a non cumulative type of resource. The analysis in [6] presents a sophisticated framework for inferring the peak heap consumption by assuming different garbage collection models. As before, in contrast to ours, the analysis is based on generating non-standard equations and for a specific type of resource. In this case, the differences are even more notable as the language in [6] is sequential. Analysis

and verification techniques of concurrent programs seek finite representations of the program traces which avoid the exponential explosion in the number of traces (see [8] and its references). In this sense, our queue configurations are a coarse representation of the traces. As future work, we plan to further improve the accuracy of our analysis by splitting tasks into fragments according to the processor release points within the task. Intuitively, if a task contains an await instruction we would divide into the code before the await and the code after. This way, we do not need to accumulate the cost of the whole task if only the fragment after the await has been queued.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

References

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPOPP*, pages 183–193. ACM, 2007.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
3. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, December 2011.
4. E. Albert, P. Arenas, S. Genaim, and D. Zanardini. Task-Level Analysis for a Language with Async-Finish parallelism. In *Proc. of LCTES'11*, pages 21–30. ACM Press, 2011.
5. 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.
6. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Garbage Collected Languages. *Science of Computer Programming*, 78(9):1427–1448, 2013.
7. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *LCPC'05*, volume 4339 of *LNCS*, pages 152–169. Springer, 2005.
8. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142. ACM, 2013.
9. 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.
10. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.
11. J. K. Lee and J. Palsberg. Featherweight x10: a core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, 2010.
12. J. K. Lee, J. Palsberg, and R. Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.

13. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.
14. M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proc. of POPL'97*, pages 1–14, Paris, France, January 1997. ACM.
15. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your Contexts Well: Understanding Object-Sensitivity. In *In Proc. of POPL'11*, pages 17–30. ACM, 2011.
16. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
17. B. Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.
18. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.