

Task-Level Analysis for a Language with `async/finish` Parallelism

Elvira Albert Puri Arenas
Samir Genaim

SIC, Complutense University of Madrid
E-28040, Madrid, Spain
{elvira,puri,samir}@clip.dia.fi.upm.es

Damiano Zanardini

DIA, Technical University of Madrid
E-28660, Boadilla del Monte, Madrid, Spain
damiano@clip.dia.fi.upm.es

Abstract

The *task level* of a program is the maximum number of tasks that can be *available* (i.e., not finished nor suspended) simultaneously during its execution for any input data. Static knowledge of the task level is of utmost importance for understanding and debugging parallel programs as well as for guiding task schedulers. We present, to the best of our knowledge, the first static analysis which infers safe and precise approximations on the task level for a language with `async-finish` parallelism. In parallel languages, `async` and `finish` are basic constructs for, respectively, spawning tasks and waiting until they terminate. They are the core of modern, parallel, distributed languages like X10. Given a (parallel) program, our analysis returns a *task-level upper bound*, i.e., a function on the program's input arguments that guarantees that the task level of the program will never exceed its value along any execution. Our analysis provides a series of useful (over)approximations, going from the total number of tasks spawned in the execution up to an accurate estimation of the task level.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: [Concurrent Programming] *Distributed programming, Parallel programming*; D.3 [Programming Languages]: [Formal Definitions and Theory]

General Terms Algorithms, Languages, Theory, Verification

Keywords Parallelism, Static Analysis, Resource Consumption, X10, Java

1. Introduction

As embedded systems increase in number, complexity, and diversity, there is an increasing need of exploiting new hardware architectures, and scaling up to multicores and distributed systems built from multicores. This brings to the embedded-systems area wide interest in developing techniques that help in understanding, optimizing, debugging, finding optimal schedulings for parallel programs. Two of the key constructs for parallelism are `async` and `finish`. The `async{s}` statement is a notation for spawning tasks: the task *s* can run in parallel with any statements following it. The

`finish{s}` statement waits for termination of *s* and of all `async` statement bodies started while executing *s*. In order to develop our analysis, we consider a Turing-complete language with a minimal syntax and a simple formal semantics. A program consists of a collection of methods that all have access to a shared array. The body of the method is a sequence of statements. Each statement can be an assignment, conditional, loop, `async`, `finish`, or method call. If we add some boilerplate syntax to a program, the result is an executable X10 program. X10 [8] is a modern, parallel, distributed language intended to be very easily accessible to Java programmers. Note that, unlike in languages based on `fork` and `join` constructs, `async-finish` programs are guaranteed to be deadlock-free [19].

Our objective is to present a clear and concise formalization of the analysis on a simple imperative language that captures the essence of standard parallelism constructs.

As our main contribution, we present a novel static analysis which infers the *task level* of a parallel program, i.e., the maximum number of *available* tasks (i.e., not finished nor suspended) which can be run simultaneously along any execution of the program (provided that sufficient computing resources are available). A starting point for our work is the observation that spawning parallel tasks can be regarded as a *resource* consumed by a program. This leads to the idea of adapting powerful techniques developed in resource analysis frameworks for *sequential* programs [5, 12, 13, 24] in order to obtain sound task-level Upper Bounds (UBs) on *parallel* programs. Such adaptation is far from trivial since, among other things, the task level of a program is not an *accumulative* resource, but rather it can increase and/or decrease along the execution. This renders direct application of cost analysis frameworks for accumulative resources (such as time, total memory consumption, etc.) unsuitable. We present our novel analysis to accurately (over-)approximate the task level of a program in the following steps, which are the main contributions of this work.

1. We first produce over-approximations of the *total* number of tasks spawned along any execution of the program. This can be done by lifting existing accumulative cost analyses developed for sequential programs to the parallel setting. The results of such analysis are sound w.r.t. any particular scheduling of tasks.
2. Secondly, we present a novel approach to approximate the *peak* (or maximum) of *live* tasks, a resource which is not accumulative. The challenge here is to come up with a technique which over-approximates the peak of live tasks among all possible states that might occur in any execution of the program.
3. As a further step, we refine the previous approach and approximate the peak of *available* tasks, i.e., we exclude those tasks which are alive but suspended, thus resulting in smaller UBs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCIES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04...\$10.00

4. Then, we show how our task-level analysis can be improved by first inferring the tasks that *survive* after the method returns, i.e., those tasks that have been created during the execution of m and can be available on return from m . This improvement requires a more complex analysis but, when doable, leads to strictly more precise bounds than those in points 2 and 3.
5. We report on a prototype implementation based on the COSTA system [4], and experimentally evaluate it on a set of simplified versions of applications from the X10 website that contain interesting parallelism patterns.

2. Motivating Examples

In this section, we introduce by means of examples the main notions that our task-level analysis over-approximates, and show the results it can produce. These examples will also illustrate the following applications of our analysis: (1) It is useful for both understanding and debugging parallel programs. For instance, our analysis can infer a task-level upper bound which is larger (or smaller) than the programmer's expectations. This can clearly help finding bugs in the program. Even more, the analysis results would be "unbounded" when an instruction which spawns new tasks is (wrongly) placed within a loop which does not terminate. (2) The results are also useful for optimizing and finding optimal deployment configurations. For example, when parallelizing the program, it is not profitable to have more processors than the inferred task level.

2.1 Total Tasks versus Live Tasks

The first example implements a parallel version of the Gauss elimination algorithm. An invocation `gaussian(n)` applies the algorithm on the matrix defined by the elements (i, j) where $0 \leq i, j \leq n-1$. It assumes that the two-dimensional array `A` is initialized with integer values.

```

1 int A [][];
2
3 void gaussian(int n) {
4   for (int k=0; k<n; k++) {
5     finish for (int j=k+1; j<n; j++) async {
6       A[k, j]=A[k, j]/A[k, k];
7     }
8     finish for (int i=k+1; i<n; i++) async {
9       for (int j=k+1; j<n; j++)
10        A[i, j]:=A[i, j]-A[i, k]*A[k, j]
11     }
12 }
13 }
```

The total number of tasks spawned by this method is quadratic on n , since at each iteration of the outer loop, each of the inner loops spawn (in the worst case) $n-1$ tasks. Note that the loop at line 11 does not spawn any task. Our analysis accurately infers that at most $1 + 2n(n-1)$ tasks will be spawned along an execution.

Due to the use of `finish` at line 5, it is ensured that before entering the loop at line 8, all asynchronous tasks spawned at line 5 are finished. Likewise, all asynchronous tasks spawned by the loop at line 8 must be finished before starting the next iteration of the outer loop. Hence, in any execution of this program, the maximum number of tasks that can be alive simultaneously (or *peak of live tasks*) corresponds to the maximum of the tasks spawned by the loops at lines 5 and 8. Our analysis precisely infers that the peak of live tasks is n .

Observe that both the total and the peak number of live tasks are useful pieces of information for the programmer. E.g., by comparing the inferred upper bounds with the programmer's expectations we might detect bugs, as explained above.

2.2 Live Tasks versus Available Tasks

The following method implements the merge-sort algorithm. It sorts the elements of the array `A` between the indexes `from` and `to`. We omit the code of `merge` and only assume that it does not spawn further tasks.

```

1 int A[];
2
3 void msort(int from, int to) {
4   if ( from < to ) {
5     mid=(from+to)/2;
6     finish {
7       async msort(from, mid);
8       async msort(mid+1, to);
9     }
10    merge(from, to, mid);
11  }
12 }
```

The total number of tasks spawned by a call `msort(from,to)` is bounded by $2 * (to - from + 1) - 2$. This upper bound is obtained by proving that, in both recursive calls, the number of elements to be sorted is decreased by half and, at each recursive call, two new tasks are spawned.

For this example, we infer that the peak of live tasks and the total number of tasks are identical. This is because the recursive calls are performed within the scope of the `finish` construct. Thus, in the worst case, all tasks can be alive simultaneously (though the current task always blocks after launching the asynchronous calls). We can improve the analysis result by proving that, at each recursive call, after spawning the two asynchronous tasks, the current process becomes *inactive* by suspending its execution until the spawned tasks terminate. With this knowledge, our analysis accurately infers that the peak of *available* tasks is at most $to - from + 1$, which is almost half of the one we obtained for live tasks.

2.3 Improving Available Tasks with Escape Information

Next example is a pre-order traversal of a binary tree where, for each node i , we spawn two tasks: `activity_a(i)` and `activity_b(i)`. We omit the code of `activity_a` and `activity_b` and only assume that they do not spawn further tasks. The binary tree is represented using the array `A`, such that the nodes at positions $2*i+1$ and $2*i+2$, respectively, are the left and right children of the node at position i . The first argument n is the depth of the tree. The method is supposed to be called with `f(n,0)`.

```

1 int A[];
2
3 void f(int n, int i) {
4   if ( n > 0 ) {
5     finish {
6       async activity_a(i);
7       async activity_b(i);
8     }
9     f(n-1, 2*i+1);
10    f(n-1, 2*i+2);
11  }
12 }
```

By accumulating all asynchronous calls spawned along the execution, our analysis generates the upper bound $2 * (2^n - 1) + 1$. As expected, the obtained bound is exponential on the depth of the tree due to the two recursive calls which traverse all nodes in the tree. For the peak of available tasks, we can greatly improve the task-level bound. In particular, we can see that the asynchronous calls in lines 6 and 7 will be finished at line 8 before the recursive calls. This means that, given a call to `f`, there are no tasks that *survive* after it returns, i.e., all tasks created during a call to `f` (directly or transitively) are terminated before the execution of the call `fin-`

ishes. The use of surviving information during our analysis allows proving that there cannot be more than 2 processes simultaneously available. From the above examples, it should become clear that an upper bound on the available tasks can be of utmost importance for finding an optimal deployment configuration. For instance, in the above example, it is not worth having more than 2 processors when executing the code.

3. A Simplified X10-like Language

We develop our analysis on a representative subset of X10 [8], a parallel language which is similar to Java in its sequential part, and relies on the `async/finish` mechanism for parallelism. From X10, we take:

- a Turing-complete core consisting of conditionals, loops, assignments and a single one-dimensional array;
- methods and method calls;
- the `async` and `finish` statements.

We omit many features from X10, including places, distributions and clocks. Indeed, our simplified language is very similar to Featherweight X10 [16] (FX10 for short), a subset of X10 which has been proposed to develop formal analyses on X10. For the sake of expressiveness, our language is richer than FX10 in that it allows input parameters in method calls (in order to handle recursion), has no restriction on conditional statements and has local variables. The treatment of object fields is similar to (and simpler than) the treatment of array accesses; details are omitted for simplicity.

3.1 The Recursive Intermediate Representation

As customary in the formalization of static analyses for realistic languages, we develop our analysis on an intermediate representation (IR) of the language which allows us to provide a clearer and more concise formalization. Similar representations are used by other static analysis tools for Java (and Java bytecode) and .NET [5, 10, 11, 21, 25]. Essentially, all these tools work by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the intermediate language (in our IR, by means of rules).

Methods in the original program are represented by one or more *procedures* in the IR. A procedure is defined by one or more *guarded rules*. The translation is as follows. Given a method, each block in its CFG is represented by means of a guarded rule. Guards state the conditions under which the corresponding block can be executed (they contain the conditions in the edges of the CFG). Each rule contains as arguments those variables that are input values to the block. When the block has more than one successor in the CFG, we just create a *continuation procedure* and a corresponding call in the rule. Blocks in the continuation will be in turn defined by means of guarded rules (with mutually exclusive conditions). As a result, all forms of iteration in the program are represented by means of *recursive* calls. The array remains as a global variable in the IR. The process of obtaining the intermediate representation from X10-like programs is completely automatic. Since it is identical as for Java programs, we will not go into the technical details of the transformation (we refer to any of [5, 10, 11, 21, 25]) but just show the intuition by means of an example.

EXAMPLE 3.1. *Fig. 1 shows the intermediate representation for the example in Sec. 2.1. This example shows an interesting aspect of the IR: loops are detected and extracted in separate procedures as described in [22]. It can be observed that within the rule `gaussian` we invoke procedure `for`, which corresponds to the `for`-loop in line 4. Similarly, when entering the remaining `for`-loops in the program,*

<pre> gaussian(n) ← k:=0, for(k, n) for(k, n) ← k ≥ n for(k, n) ← k < n, j:=k+1, finish{for₁(k, n, j)}, i:=k+1, finish{for₂(k, n, i)}, k':=k+1, for(k', n) for₁(k, n, j) ← j ≥ n for₁(k, n, j) ← j < n, async{op₁(k, j)} j':=j+1, for₁(k, n, j') </pre>	<pre> for₂(k, n, i) ← i ≥ n for₂(k, n, i) ← i < n, j:=k+1 async{for_{2.1}(k, n, j, i)}, i':=i+1, for₂(k, n, i') for_{2.1}(k, n, j, i) ← j ≥ n for_{2.1}(k, n, j, i) ← j < n, op₂(k, j, i), j':=j+1, for_{2.1}(k, n, j', i) </pre>
---	---

Figure 1. Recursive Intermediate Representation of Ex. in Sec. 2.1

we have calls in the IR to corresponding procedures defining them. By looking at the two rules defining procedure `for`, we observe the more interesting aspects of our IR: (1) rules contain as arguments those variables that are input values to the scope of the loop; (2) by means of guards, we distinguish the case of exiting the loop (first rule) and entering the loop (second rule); (3) iteration is transformed into recursion. Note that, in the IR, the `finish` construct is applied on a single instruction. If there are several instructions within the scope of `finish` in the original program, we just create an auxiliary procedure which contains them all.

3.2 Syntax

A *program* in our intermediate representation consists of a set of *procedures*, each of them defined by one or more *guarded rules*. In the following, given any entity a , we use \bar{a} to denote the sequence a_1, \dots, a_n , $n \geq 1$. A procedure p with k input arguments \bar{x} is defined by rules which adhere to this grammar:

$$\begin{aligned}
\text{rule} &::= p(\bar{x}) \leftarrow g, b, \dots, b \\
g &::= \text{true} \mid \text{exp op exp} \\
b &::= y := \text{exp} \mid \mathbf{A}[y] := \text{exp} \mid q(\bar{x}) \mid \text{async}\{q(\bar{x})\} \mid \text{finish}\{q(\bar{x})\} \\
\text{exp} &::= y \mid d \mid \mathbf{A}[y] \mid \text{exp} - \text{exp} \mid \text{exp} + \text{exp} \mid \text{exp} * \text{exp} \mid \text{exp} / \text{exp} \\
\text{op} &::= > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{aligned}$$

where $p(\bar{x})$ is the *head* of the rule; g is its guard, which specifies conditions for the rule to be applicable; the sequence b, \dots, b is the body of the rule; d is an integer; y is a variable name; $q(\bar{x})$ is a procedure call, $\text{async}\{q(\bar{x})\}$ is an asynchronous procedure call, and $\text{finish}\{q(\bar{x})\}$ is a synchronized call. All variables are of type integer. Computations work on a single shared memory given by a one-dimensional array of integer values named \mathbf{A} with indexes $0 \dots N-1$. When the execution begins, input values are loaded into all elements of the array. Thus, the array \mathbf{A} is fully initialized for all indices $0 \dots N-1$. In the examples, to simplify the presentation, we use several (possibly multidimensional) arrays.

3.3 Semantics

Fig. 2 shows the *operational semantics* for X10-like programs in the IR. It adapts the small-step operational semantics of FX10 [16] to our syntax and extends it to handle the additional language features discussed in the beginning of the section. It uses the binary operator \parallel in the semantics of `async` and \triangleright in `finish`. A *state* is a pair, consisting of the state of the array and a tree which describes the code executing. Namely, it is of the form $(\mathbf{A}; T)$ where $\mathbf{A} : \{0, \dots, N-1\} \mapsto \mathbb{Z}$ is an array of integers and T is an execution tree defined by the following grammar:

$$T ::= T \triangleright T \mid T \parallel T \mid \langle id, instr, tv \rangle$$

where $id \in \mathbb{N}$ is a unique task identifier, $instr$ is a sequence of instructions (as in Sec. 3.2) and $tv: \mathcal{V} \mapsto \mathbb{Z}$ is a partial map from the set of variable names \mathcal{V} to integers. The symbol ϵ denotes an empty sequence of instructions. We refer to the tuple $\langle id, \bar{b}, tv \rangle$ as a record.

$$\begin{array}{l}
(1) \quad \frac{}{(\mathbf{A}; \langle id, \epsilon, tv \rangle \triangleright T) \rightarrow (\mathbf{A}; T)} \\
(2) \quad \frac{(\mathbf{A}; T_1) \rightarrow (\mathbf{A}'; T_1')}{(\mathbf{A}; T_1 \triangleright T_2) \rightarrow (\mathbf{A}'; T_1' \triangleright T_2)} \\
(3) \quad \frac{}{(\mathbf{A}; \langle id, \epsilon, tv \rangle \parallel T) \rightarrow (\mathbf{A}; T)} \\
(4) \quad \frac{}{(\mathbf{A}; T \parallel \langle id, \epsilon, tv \rangle) \rightarrow (\mathbf{A}; T)} \\
(5) \quad \frac{(\mathbf{A}; T_1) \rightarrow (\mathbf{A}'; T_1')}{(\mathbf{A}; T_1 \parallel T_2) \rightarrow (\mathbf{A}'; T_1' \parallel T_2)} \\
(6) \quad \frac{(\mathbf{A}; T_2) \rightarrow (\mathbf{A}'; T_2')}{(\mathbf{A}; T_1 \parallel T_2) \rightarrow (\mathbf{A}'; T_1 \parallel T_2')} \\
(7) \quad \frac{b \equiv x := exp, \quad v = eval(exp, tv, \mathbf{A})}{(\mathbf{A}; \langle id, b \cdot instr, tv \rangle) \rightarrow (\mathbf{A}; \langle id, instr, tv[x \mapsto v] \rangle)} \\
(8) \quad \frac{b \equiv \mathbf{A}[x] := exp, \quad v = eval(exp, tv, \mathbf{A}), \quad n = tv(x), \quad 0 \leq n \leq \mathbf{N} - 1}{(\mathbf{A}; \langle id, b \cdot instr, tv \rangle) \rightarrow (\mathbf{A}[n \mapsto v]; \langle id, instr, tv \rangle)} \\
(9) \quad \frac{b \equiv q(\bar{x}), \quad r \equiv q(\bar{x}') \leftarrow g, b_1, \dots, b_k \ll_{tv} P, \quad eval(g, tv', \mathbf{A}) \equiv true}{(\mathbf{A}; \langle id, b \cdot instr, tv \rangle) \rightarrow (\mathbf{A}; \langle id, b_1 \cdot \dots \cdot b_k \cdot instr, tv' \rangle)} \\
(10) \quad \frac{b \equiv \mathbf{async}\{q(\bar{x})\}, \quad id' \text{ is a new identifier not used before}}{(\mathbf{A}; \langle id, b \cdot instr, tv \rangle) \rightarrow (\mathbf{A}; \langle id', q(\bar{x}), tv \rangle \parallel \langle id, instr, tv \rangle)} \\
(11) \quad \frac{b \equiv \mathbf{finish}\{q(\bar{x})\}}{(\mathbf{A}; \langle id, b \cdot instr, tv \rangle) \rightarrow (\mathbf{A}; \langle id, q(\bar{x}), tv \rangle \triangleright \langle id, instr, tv \rangle)}
\end{array}$$

Figure 2. Operational semantics

Executions start from an *initial state* of the form $(\mathbf{A}; \langle 1, p(\bar{x}), tv \rangle)$, where p is the entry procedure name, and the elements of the array \mathbf{A} and $tv(x_i)$ for all $x_i \in \bar{x}$ are initialized to some initial values. We often view tv as a set $\{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$ where each x_i is a variable name and each v_i is an integer value. Executions are regarded as *traces* of the form $(\mathbf{A}_0; T_0) \rightarrow (\mathbf{A}_1; T_1) \rightarrow \dots \rightarrow (\mathbf{A}_n; T_n)$, sometimes denoted as $(\mathbf{A}_0; T_0) \rightarrow^* (\mathbf{A}_n; T_n)$. Infinite traces correspond to non-terminating executions. We say that a call to a procedure *locally* terminates if the execution of its procedure's body terminates, and we say that it *globally* terminates if, in addition, all tasks it spawns terminate.

The left side of Fig. 2 contains the rules for dealing with parallelism and synchronization. A tree $T_1 \triangleright T_2$ gives the semantics of the `finish` statement. As shown in rule (2), T_1 must complete execution before moving on to executing T_2 , i.e., T_1 must be reduced to $\langle id, \epsilon, tv \rangle$ in order to apply rule (1). Rules (3) and (4) remove trees whose evaluation is completely finished whereas (5) and (6) allow choosing trees T_1 or T_2 non-deterministically (i.e., there is no assumption on the task scheduler).

The right side of Fig. 2 contains the rules for executing instructions. Intuitively, rule (7) accounts for all instructions in the semantics which perform arithmetic and assignment operations. We assume that $eval(exp, tv, \mathbf{A})$ returns the evaluation of the arithmetic expression exp using the values of the corresponding variables from tv and \mathbf{A} in the standard way. Moreover, we assume that it fails when trying to access \mathbf{A} with an index which is not in the range $0 \dots \mathbf{N} - 1$. Rule (8) deals with assignments on \mathbf{A} . After evaluating exp , the resulting value is stored in the position $tv(x)$ of \mathbf{A} . Rule (9) corresponds to invoking a procedure $q(\bar{x})$. It first takes a rule r for q . The notation \ll_{tv} means that we rename the rule variables so they will not clash with names already in the domain of tv . Then, we generate a new variable mapping tv' which extends tv by initializing the formal parameters \bar{x}' with the values of the actual parameters \bar{x} , and the remaining variables not in \bar{x} (i.e., $vars(r) \setminus \bar{x}$) to 0. We require that the guard g of rule r is evaluated to *true* (as usual, the values *true* and *false* can be simulated with 0 and non-0 integers). Rule (10) takes care of the `async` statement by spawning a new task to be executed in parallel. Finally, rule (11) introduces the operator \triangleright to wait for the termination of the task, when we have a `finish` instruction.

EXAMPLE 3.2. *As an example of how the semantics works, consider the following simple program. For brevity, we ignore the code of procedures q_1, \dots, q_5 and assume that they neither make directly or indirectly any asynchronous call, nor modify the array.*

$p \leftarrow \mathbf{async}\{q_1\}, \mathbf{finish}\{q\}, \circlearrowleft \mathbf{async}\{q_2\}, q_3$
 $q \leftarrow \mathbf{async}\{q_4\}, \mathbf{async}\{q_5\}$

The following derivation starts from the entry procedure p :

$$\begin{array}{l}
(\mathbf{A}; \langle 1, p, tv \rangle) \rightarrow \\
(\mathbf{A}; \langle 1, \mathbf{async}\{q_1\} \cdot \mathbf{finish}\{q\} \cdot \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
*_1 (\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \langle 1, \mathbf{finish}\{q\} \cdot \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
*_2 (\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \langle 1, q, tv \rangle \triangleright \langle 1, \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
(\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \\
\langle 1, \mathbf{async}\{q_4\} \cdot \mathbf{async}\{q_5\}, tv \rangle \triangleright \langle 1, \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
(\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \\
\langle 3, q_4, tv \rangle \parallel \langle 1, \mathbf{async}\{q_5\}, tv \rangle) \triangleright \langle 1, \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
\diamond (\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \\
\langle 3, q_4, tv \rangle \parallel \langle 4, q_5, tv \rangle \parallel \langle 1, \epsilon, tv \rangle) \triangleright \langle 1, \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow^* \\
(\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \langle 1, \epsilon, tv \rangle) \triangleright \langle 1, \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
*_3 (\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \langle 1, \mathbf{async}\{q_2\} \cdot q_3, tv \rangle) \rightarrow \\
(\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \langle 5, q_2, tv \rangle \parallel \langle 1, q_3, tv \rangle) \rightarrow \\
(\mathbf{A}; \langle 2, q_1, tv \rangle \parallel \langle 5, q_2, tv \rangle \parallel \langle 1, \epsilon, tv \rangle) \rightarrow^* (\mathbf{A}; \langle 2, \epsilon, tv \rangle)
\end{array}$$

Note that since q_1 is invoked asynchronously, p can continue to the next statement at $*_1$. However, when executing `finish`{ q } at $*_2$, the execution of p blocks until q and its asynchronous sub-tasks q_4 and q_5 terminate, then resumes from the program point \circlearrowleft (step $*_3$).

4. Basic Definitions in Task Parallelism

We first introduce basic notions related to the task parallelism of a program. They define the notions that later we want to approximate by means of static analysis. First, we introduce two auxiliary definitions to count the number of tasks that can be simultaneously alive at some program point by means of the following function $live(T)$ which goes from the set of trees to the set of task identifiers $\wp(\mathbb{N})$:

$$\begin{array}{l}
live(T_1 \parallel T_2) = live(T_1) \cup live(T_2) \\
live(T_1 \triangleright T_2) = live(T_1) \cup live(T_2) \\
live(\langle id, \epsilon, tv \rangle) = \emptyset \\
live(\langle id, instr, tv \rangle) = \{id\}
\end{array}$$

Note that when a task does not have any further instruction to execute (third equation) it is not counted as alive. The above definition includes tasks which are *blocked*, i.e., are not available in the current state. For instance, for $(S_1 \parallel S_2) \triangleright S_3$ such that each S_i has $live(S_i) = 1$, the function $live$ returns 3. However, the semantics of \triangleright ensures that S_3 is blocked, i.e., it remains suspended until the execution of $S_1 \parallel S_2$ finishes. The function available counts only the available tasks, i.e., live tasks which are not suspended. It is defined as $live$ except for $available(T_1 \triangleright T_2) = available(T_1)$. Given a trace $t \equiv T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_n$, by relying on the above two functions, we can define the following three important notions that our analysis approximates:

- $\text{total}(t)$. First, we define the *total number of spawned tasks* along an execution, which corresponds to the total number of tasks that have been started, as: $\text{total}(t) = |\cup_{i=0}^n \text{live}(T_i)|$ (here, $|X|$ is the size of the set X). Note that this resource is accumulative, i.e., it always increases as the execution proceeds.
- $\text{peakLive}(t)$. Another interesting notion is the peak of *live tasks*, i.e., the maximum number of tasks that are simultaneously started and not finished. The function $\text{peakLive}(t)$ is defined as $\max(\{|\text{live}(T_0)|, \dots, |\text{live}(T_n)|\})$. Note that this resource is not accumulative: instead, the number of live tasks can increase or decrease at any state. Thus, in order to approximate it, we need to observe all states and capture the maximum.
- $\text{peakAvailable}(t)$. Similarly, we can define the peak of *available tasks* along the execution, i.e., the maximum number of tasks that are simultaneously started and not blocked. Thus, $\text{peakAvailable}(t) = \max(\{|\text{available}(T_0)|, \dots, |\text{available}(T_n)|\})$. We also refer to this notion as the *task level* of the execution, the two definitions above being over-approximations of it.

EXAMPLE 4.1. By applying the above definitions to the derivation of Ex. 3.2, we have: $\text{total}(t) = 5$, $\text{peakLive}(t) = 4$ and $\text{peakAvailable}(t) = 3$. Note that the difference between peakLive and peakAvailable occurs in the state labeled \diamond . This is because, after creating the new tasks, the task on which q is executing is alive but blocked (hence not available).

5. Static Inference of Total Spawned Tasks

In the previous section, definitions assume a specific trace. Thus, the program must be executed on a concrete input in order to compute the functions. Now, we want to approximate these notions *statically*, i.e., without executing the program, so that the results will be valid for any input. In particular, by concentrating on the total number of spawned tasks first, given $p(\bar{x})$, the goal is to infer $p^{ub}(\bar{x})$, called *task-level upper bound for p* , which is a function on input data guaranteeing that, given any concrete values \bar{v} for \bar{x} , the total number $\text{total}(t)$ of tasks spawned along the trace t of $p(\bar{v})$ not greater than $p^{ub}(\bar{v}) + 1$ (1 corresponds to the main task).

Since the total number of tasks is an accumulative resource, in principle, any of the existing resource analysis frameworks that count a particular form of accumulative resource (e.g., instructions [12], total memory [13, 24], etc.) can be adapted to the total number of spawned tasks by counting the instructions `async` that spawn tasks and ignoring the rest. However, all above approaches deal with sequential programs, and should be lifted to the parallel setting. This is because, as we will see later, the resulting UB can be affected by the fact that tasks can run in parallel. Among all possible resource analysis frameworks, we rely on the most traditional one, proposed by Wegbreit [27] in 1975. As our first contribution, we adapt such approach to infer sound results on the task level in a parallel setting. The next three subsections present the main steps of the analysis:

- First, we discuss in Sec. 5.1 the value abstraction component which is used to infer inter-relations between the program variables. Interestingly, by losing information about the global data during the value abstraction, we are able to ensure soundness of the overall UBs in the parallel setting.
- Given the value relations, we proceed in Sec. 5.2 to define, for our intermediate language, how to generate the recurrence equations which define the spawned tasks.
- Finally, in Sec. 5.3, we briefly describe the process of obtaining safe over-approximations from the generated recurrence equations by relying on existing solvers.

5.1 Value Abstraction

Given a rule, we describe how to generate a conjunction of (linear) constraints (sometimes written as a set) that describes the relations between the values of the rule’s variables at the different program points. This information is later used, for example, to understand how values change when moving from one procedure to another. In particular, it is essential for bounding the number of recursive calls (i.e., iterations of loops). The following definition presents the notion of *value abstraction* for a given rule. In order to distinguish between the values of a variable at different program points (inside a single rule), rules are given in static single assignment (SSA) form [6] (array accesses remain the same). The rules in Fig. 1 and in all remaining examples are in SSA. This transformation is straightforward for a single rule, as rules do not have branching.

DEFINITION 5.1. Give a rule $r \equiv p(\bar{x}) \leftarrow g, b_1 \dots, b_n$ in SSA form, its value abstraction is $\varphi_r = \alpha(g) \wedge \alpha(b_1) \wedge \dots \wedge \alpha(b_n)$ where:

- $\alpha(y := \text{exp}) = (y = \text{exp})$ if exp is a linear expression which does not involve arrays;
- $\alpha(\text{exp}_1 \text{ op } \text{exp}_2) = (\text{exp}_1 \text{ op } \text{exp}_2)$ if $\text{op} \in \{>, \geq, <, \leq, =\}$ and exp_1 and exp_2 are linear expressions not involving arrays;
- $\alpha(b) = \text{true}$, otherwise.

For simplicity, the above abstraction ignores non-linear arithmetic expressions by abstracting the corresponding instructions to *unknown (true)*. Non-linear arithmetic can be handled at the price of performance using non-linear constraints manipulation techniques.

EXAMPLE 5.2. Applying Def. 5.1 on the second rule for “for” of Fig. 1, we obtain as value abstraction $\{k < n, k' = k + 1, j = k', i = k'\}$.

An important point in the above abstraction is that data in the global array A are ignored. This provides us correctness in the context of parallel execution without requiring any other sophisticated heap analysis for ensuring the *independence* [26] between tasks.

EXAMPLE 5.3. Consider the following program and observe that when m invokes the two asynchronous calls, procedures p and q might run in parallel depending on the underlying task scheduler.

```

m(n) ← async{p(0, n)}, async{q(n)}
p(i, n) ← i ≥ A[n]
p(i, n) ← i < A[n], async{q1}, i' := i + 1, p(i', n)
q(n) ← A[n] := A[n] + 1, q(n)

```

By looking at a complete execution of p in isolation (i.e., if it does not interleave with that of q), we can see that a sound upper bound on the number of tasks spawned by p is $A[n]$ (the value of the n -th element of the array). However, if the execution of q interleaves with that of p , the execution of p might not terminate since q increases the value of $A[n]$. Hence, the previous UB is not correct.

Our practical solution to avoid the above problem is to abstract instructions that involve global data (i.e., array elements) to *unknown* (i.e., *true*). In the above case, the guard $i < A[n]$ is abstracted to *true*, so that the value of $A[n]$ is lost. Hence, we will not be able to infer an UB for the method. This does not mean that we cannot analyze programs that use the array but rather that, when the UB is a function of an array element, we cannot find it. In Sec. 11, we discuss how to improve the accuracy by relying on a may-happen-in-parallel analysis [16] in combination with a field-sensitive value analysis [17]. It should be noted that the value abstraction is an independent component in our analysis and we can improve it regardless of the next components that we will introduce in what follows. Also, when improving it, we can integrate advanced value abstractions for data structures such as path-length [21] or term value [18], without any modification to the rest of our analysis.

5.2 Generation of Recurrence Equations

Given a program P and the value abstractions of its rules, a recurrence-relation (RR) system for P is generated by applying the following definition to all rules in the program.

DEFINITION 5.4 (total number of spawned tasks). *Let r be a rule of the form $p(\bar{x}) \leftarrow g, b_1, \dots, b_n$, and φ_r be its corresponding value relations as computed in Def. 5.1. Then, its total-tasks equation is defined as $p(\bar{x}) = \sum_{i=1}^n \mathcal{T}(b_i), \varphi_r$, where*

$$\begin{aligned} \mathcal{T}(b) &= 1 + q(\bar{x}) && \text{if } b = \text{async}\{q(\bar{x})\} \\ \mathcal{T}(b) &= q(\bar{x}) && \text{if } b = \text{finish}\{q(\bar{x})\} \\ \mathcal{T}(b) &= q(\bar{x}) && \text{if } b = q(\bar{x}) \\ \mathcal{T}(b) &= 0 && \text{otherwise} \end{aligned}$$

The set of equations generated for a program P is denoted by \mathcal{S}_P .

EXAMPLE 5.5. *By applying the above definition to the rules of Fig. 1, we obtain the following set of total tasks equations:*

$$\begin{aligned} \text{gaussian}(n) &= \text{for}(k, n) && \{k=0\} \\ \text{for}(k, n) &= 0 && \{k \geq n\} \\ \text{for}(k, n) &= && \{k < n, k' = k + 1, \\ & \text{for}_1(n, j) + \text{for}_2(k, n, i) + \text{for}(k', n) && j = k', i = k'\} \\ \text{for}_1(n, j) &= 0 && \{j \geq n\} \\ \text{for}_1(n, j) &= 1 + \text{for}_1(n, j') && \{j < n, j' = j + 1\} \\ \text{for}_2(k, n, i) &= 0 && \{i \geq n\} \\ \text{for}_2(k, n, i) &= && \{i < n, i' = i + 1, \\ & 1 + \text{for}_{2.1}(n, j) + \text{for}_2(k, n, i') && j = k + 1\} \\ \text{for}_{2.1}(n, j) &= 0 && \{j \geq n\} \\ \text{for}_{2.1}(n, j) &= \text{for}_{2.1}(n, j') && \{j < n, j' = j + 1\} \end{aligned}$$

The only rules in Fig. 1 which contain `async` are the second ones in `for1` and `for2`. Their corresponding equations accumulate 1 for such instruction. Value relations are transformed into linear constraints attached to the equations. They contain the applicability conditions for the rules and how the values of variables change when moving from one procedure to another.

5.3 Closed-form Upper Bounds

Once the RR are generated, a worst-case cost analyzer uses a solver in order to obtain closed-form UBs, i.e., *cost expressions* without recurrences. Traditionally, cost analyzers rely on computer algebra systems (e.g., MAXIMA, MAPLE) to solve the obtained recurrences. As other advanced systems [23], ours includes a dedicated solver, called PUBS [2], in order to be able to handle more types of RR. The technical details of the process of obtaining a cost expression from the RR are not explained in the paper, as the presented analysis does not require any adaptation for this part in the general framework. Given a RR $p(\bar{x})$, we denote by $p^{ub}(\bar{x})$ its closed-form UB, which is a cost expression of the following form (and could be obtained by any of the above solvers):

$$e \equiv q|\text{nat}(l)| \log(\text{nat}(l) + 1) |e * e| e + e | 2^{\text{nat}(l)} | \max(e, \dots, e)$$

where q is positive rational number, l is a linear expression, and function `nat` is defined as $\text{nat}(v) = \max\{v, 0\}$.

EXAMPLE 5.6. *As usual, UBs are obtained by first computing UBs for cost relations which do not depend on any other relation and continuing by replacing the computed UBs on the equations which call such relations. The solutions for the equations in Ex. 5.5 are:*

$$\begin{aligned} \text{for}_{2.1}(n, j) &= 0 \in O(1) \\ \text{for}_2(k, n, i) &= n - i \in O(n - i) \\ \text{for}_1(n, j) &= n - j \in O(n - j) \\ \text{for}(k, n) &= 2(n - k)(n - k - 1) \in O((n - k)^2) \\ \text{gaussian}(n) &= 2n(n - 1) \in O(n^2) \end{aligned}$$

As intuitively explained in Sec. 2.1, the UB we obtain for the method `gaussian` is quadratic on n . We will add 1 to this UB in order to count the task in which the initial call `gaussian(n)` is executing.

The following theorem states the soundness of our total tasks analysis. Intuitively, the main issue is to prove that derivations in the equations of Def. 5.4 capture all possible paths in a parallel execution of the program (and, due to the over-approximation in the value abstraction, possibly more). We assume soundness of the UBs solver. In the following theorems, 1 is added to the UB in order to count the current task on which the initial call is executed.

THEOREM 5.7. *Let P be a program with an entry procedure p , and let $p^{ub}(\bar{x})$ be a closed-form UB function for $p(\bar{x}) \in \mathcal{S}_P$. Then, for any trace $t \equiv (\mathbf{A}_0; \langle 1, p(\bar{x}), tv \rangle) \rightarrow^* (\mathbf{A}_n; T_n)$, it holds that $p^{ub}(\bar{v}) + 1 \geq \text{total}(t)$, where $v = tv(\bar{x})$.*

6. Inference of Peak of Live Tasks

The previous section over-approximated total, which is an accumulative resource (Sec. 4). In this section, our goal is to over-approximate live, a non-accumulative resource that might increase and/or decrease along execution. The main difference is that, in accumulative resources, one can just consider for the over-approximation the resource consumption in the final state. This is what traditional RR (like those in Def. 5.4) do. However, in the case of non-accumulative resources, one aims at observing all states where the consumption can be maximal, not only the final one. For our particular resource (the task level), an important observation is that it is enough to approximate the behavior of the program around the program points in which the number of tasks can decrease, i.e., when reaching a `finish`. Such points can be detected syntactically from the program. The key idea of our analysis is to introduce a *disjunction* between the task level just before executing each `finish` and the task level reached after the `finish` resumes execution. The peak is the maximum of both disjuncts.

EXAMPLE 6.1. *Consider again Ex. 3.2. The peak of live tasks can be found as the maximum of the following two scenarios:*

1. peak before `finish`{ q } globally terminates: 1 for `async`{ q_1 }, plus the peak of live tasks of q (which is 2); and
2. peak after `finish` is executed: 1 for `async`{ q_1 }, since it might still be alive at $\textcircled{1}$, plus 1 for `async`{ q_2 } and 0 for q_3 .

Note that, in scenario 2, we do not count the tasks created during the execution of q since `finish` guarantees that they are not alive when we reach program point $\textcircled{1}$. In summary, the peak of live tasks when executing p is 3. Additionally, we add 1 for the task in which p is running. This coincides what we have obtained in Ex. 4.1 for a particular trace.

Next definition presents a novel form of RR, called *peak-live equations*, which over-approximates the peak of live tasks along any execution of the program, according to the above intuition.

DEFINITION 6.2 (peak-live equations). *Let r be a rule $p(\bar{x}) \leftarrow g, b_1, \dots, b_n$ in SSA form and φ_r its corresponding value abstraction. Then, its equation for the peak of live tasks is $\hat{p}(\bar{x}) = \mathcal{P}(b_1, \dots, b_n), \varphi_r$, where \mathcal{P} is defined recursively as follows:*

$$\begin{aligned} \mathcal{P}(\epsilon) &= 0 \\ \mathcal{P}(b \cdot \text{instr}) &= 1 + \hat{q}(\bar{z}) + \mathcal{P}(\text{instr}) && \text{if } b = \text{async}\{q(\bar{z})\} \\ \mathcal{P}(b \cdot \text{instr}) &= \max(\hat{q}(\bar{z}), \mathcal{P}(\text{instr})) && \text{if } b = \text{finish}\{q(\bar{z})\} \\ \mathcal{P}(b \cdot \text{instr}) &= \hat{q}(\bar{z}) + \mathcal{P}(\text{instr}) && \text{if } b = q(\bar{z}) \\ \mathcal{P}(b \cdot \text{instr}) &= \mathcal{P}(\text{instr}) && \text{otherwise} \end{aligned}$$

The set of equations generated for a program P is denoted by $\hat{\mathcal{S}}_P$.

Intuitively, in the above definition, we transform the peak of tasks for a given (non-empty) sequence of instructions by transforming each instruction as follows: (i) when we find an `async`{ $q(\bar{x})$ } statement, we count one new task plus the peak of tasks created along the execution of $q(\bar{x})$; (ii) in the case of `finish`{ $q(\bar{x})$ }, since

it is ensured that all tasks created during the execution of $q(\bar{x})$ are terminated, we take the maximum between the peak reached during the execution of $q(\bar{x})$ and the peak reached after executing the $\text{finish}\{q(\bar{x})\}$; (iii) when we find a method call, we accumulate the peak reached during its execution with the continuation; and (iv) the remaining instructions are ignored.

EXAMPLE 6.3. Let us first see the equations generated for the simple program of Ex. 6.1. Note that, as there are no variables, all φ_r are simply true and can be ignored.

$$\begin{aligned}\hat{p} &= 1 + \hat{q}_1 + \max(\hat{q}, 1 + \hat{q}_2 + \hat{q}_3) \\ \hat{q} &= 1 + \hat{q}_4 + 1 + \hat{q}_5\end{aligned}$$

In order to solve the above recurrence equations, the max operator can be eliminated by transforming the equation into several non-deterministic equations, e.g., $\hat{p}(\bar{x}) = A + \max(B, C)$, φ is translated into the two equations $\hat{p}(\bar{x}) = A + B, \varphi$ and $\hat{p}(\bar{x}) = A + C, \varphi$. Solving the above equations, under the assumption that $\hat{q}_i = 0$ for all $1 \leq i \leq 5$, results in $\hat{q} = 2$ and $\hat{p} = 3$. In this example, the accuracy gain of live w.r.t. total is just constant but, in general, it can be much larger. For instance, the peak live equations for the example in Sec. 2.1 are:

$$\begin{aligned}\text{gaussian}(n) &= \hat{\text{for}}(k, n) && \{k=0\} \\ \hat{\text{for}}(k, n) &= 0 && \{k \geq n\} \\ \hat{\text{for}}(k, n) &= \max\{\hat{\text{for}}_1(n, j), \max\{\hat{\text{for}}_2(k, n, i), \hat{\text{for}}(k', n)\}\} && \{k < n, k' = k+1, j = k', i = k'\} \\ \hat{\text{for}}_1(n, j) &= 0 && \{j \geq n\} \\ \hat{\text{for}}_1(n, j) &= 1 + \hat{\text{for}}_1(n, j') && \{j < n, j' = j+1\} \\ \hat{\text{for}}_2(k, n, i) &= 0 && \{i \geq n\} \\ \hat{\text{for}}_2(k, n, i) &= 1 + \hat{\text{for}}_2(k, n, i') && \{i < n, i' = i+1, j = k+1\}\end{aligned}$$

The solution of $\hat{\text{for}}_1$ and $\hat{\text{for}}_2$ is like in Ex. 5.6. After replacing them in the second equation of $\hat{\text{for}}$ and eliminating max, we obtain the peak live UB $\text{gaussian}(n) = n - 1 \in O(n)$. Note that the total UB was quadratic on n . Again, we should add 1 to count the task in which the initial call is being executed.

The following theorem states that the solutions of the equations generated in Def. 6.2 is a sound approximation of peakLive.

THEOREM 6.4. Let P be a program with an entry procedure p , and let $\hat{p}^{\text{ub}}(\bar{x})$ be a closed-form UB function $\hat{p}(\bar{x}) \in \hat{S}_P$. Then, for any trace $t \equiv (A_0 ; \langle 1, p(\bar{x}), tv \rangle) \rightarrow^* (A_n ; T_n)$ it holds that $\hat{p}^{\text{ub}}(\bar{v}) + 1 \geq \text{peakLive}(t)$ where $\bar{v} = tv(\bar{v})$.

7. Inference of Peak of Available Tasks

The goal of this section is to accurately approximate peakAvailable, or the task level properly said. Note that, when inferring peakLive in the previous section, we have possibly included tasks which are alive but suspended. For the applications discussed in Sec. 2, it is clearly useful to exclude suspended tasks from the peak: e.g., it is not worth allocating suspended tasks in a separate processor.

EXAMPLE 7.1. Consider again the program of Ex. 6.1, and recall that, in Ex. 6.3, we inferred that the peak of live tasks is $\hat{p} = 3$ (plus 1 for the task in which p is running). However, during the execution of p , the maximum number of tasks which are available (not suspended) is just 3. This is because the task in which p is executing is only available until it reaches the call $\text{async}\{q_5\}$; as soon as q_5 is invoked asynchronously, p suspends and has to wait for q_4 and q_5 to terminate before proceeding to program point $\textcircled{1}$.

In general, it is not easy to detect when tasks are blocked, since the execution of $\text{finish}\{p(\bar{x})\}$ often spawns asynchronous calls but also executes other instructions. Therefore, the task in which $\text{finish}\{p(\bar{x})\}$ is executed does not always block. However, in

cases where the last instruction of $p(\bar{x})$ (directly or indirectly) is an asynchronous call, we have a behavior similar to the above example, i.e., at the same time the task in which $\text{finish}\{p(\bar{x})\}$ is executing suspends and another task starts. Many of these cases can be syntactically detected and treated in a special way. In what follows, we explain how to handle a common pattern in which $p(\bar{x})$ consists of only asynchronous calls, as in the above example. In order to keep the task-level analysis as simple as possible, we introduce an auxiliary construct in the language, called **finish-async**, by means of the following program transformation.

DEFINITION 7.2 (**finish-async**). Given an instruction of the form $\text{finish}\{p(\bar{x})\}$, if p is defined by a single rule of the form $p(\bar{x}) \leftarrow \text{async}\{q_1(\bar{x}_1)\}, \dots, \text{async}\{q_n(\bar{x}_n)\}$, then we replace the original instruction by **finish-async** $\{q_1(\bar{x}_1), \dots, q_n(\bar{x}_n)\}$.

Well-known transformations such as *unfolding* can be used to detect the above pattern in the presence of intermediate rules, so that the transformation can be applied more often. For instance, if we have, $p \leftarrow q, \dots, \text{async}\{q_n\}$ where q is defined as $q \leftarrow \text{async}\{q_1\}$, we need to unfold the body of q in order to be able to introduce the **finish-async** construct. Luckily, this is a well-studied problem in the field of partial evaluation [15], and existing unfolding strategies can be directly applied to our context.

DEFINITION 7.3 (peak-available equations). The peak-available equations extend those of Def. 6.2 with the additional case

$$\mathcal{P}(b \cdot \text{instr}) = \max(n-1 + \hat{q}_1(\bar{z}_1) + \dots + \hat{q}_n(\bar{z}_n), \mathcal{P}(\text{instr}))$$

which is applied when $b = \text{finish-async}\{q_1(\bar{z}_1), \dots, q_n(\bar{z}_n)\}$.

EXAMPLE 7.4. Applying the **finish-async** transformation on the program of Ex. 3.2 results in the following rule for p

$$p \leftarrow \text{async}\{q_1\}, \text{finish-async}\{q_4, q_5\}, \text{async}\{q_2\}, q_3$$

Applying Def. 7.3, we obtain the following peak-available equation: $\hat{p} = 1 + \hat{q}_1 + \max(1 + \hat{q}_4 + \hat{q}_5, 1 + \hat{q}_2 + \hat{q}_3)$. Solving the above equation, under the assumption that $\hat{q}_i = 0$ for all $1 \leq i \leq 5$, results in $\hat{p} = 2$. Therefore, at most $\hat{p}+1 = 3$ tasks might be available at the same time. The improvement achieved by the peak-available equations w.r.t. the live ones can be significant. For instance, consider the IR for program in Sec. 2.2:

$$\begin{aligned}\text{msort}(\text{from}, \text{to}) &\leftarrow \text{from} \geq \text{to}. \\ \text{msort}(\text{from}, \text{to}) &\leftarrow \text{from} < \text{to}, \text{mid} := (\text{from} + \text{to}) / 2, \\ &\quad \text{finish-async}\{\text{msort}(\text{from}, \text{mid}), \text{msort}(\text{mid} + 1, \text{to})\} \\ &\quad \text{merge}(\text{from}, \text{to}, \text{mid}).\end{aligned}$$

We show at the top (resp., bottom) the equations obtained by applying Def. 6.2 (resp., Defs. 7.2 and 7.3) to the above rules:

$\begin{aligned}\hat{\text{msort}}(f, t) &= 0 \quad \{f \geq t\} \\ \hat{\text{msort}}(f, t) &= \max(\hat{a}ux(f, t, m'), \hat{\text{merge}}(f, t, m')) \\ &\quad \{f < t, 2m' = f + t\} \\ \hat{a}ux(f, t, m) &= 2 + \hat{\text{msort}}(f, m) + \hat{\text{msort}}(m', t) \quad \{m' = m + 1\}\end{aligned}$
$\begin{aligned}\hat{\text{msort}}(f, t) &= 0 \quad \{f \geq t\} \\ \hat{\text{msort}}(f, t) &= \\ &\quad \max(1 + \hat{\text{msort}}(f, m') + \hat{\text{msort}}(m'', t), \hat{\text{merge}}(f, t, m')) \\ &\quad \{f < t, 2m' = f + t, m'' = m' + 1\}\end{aligned}$

As pointed out in Sec. 2.2, the solution for the equations at the top is $2 * (t - f + 1) - 2$, while for the ones at the bottom is $(t - f + 1)$. Clearly, available tasks are a most useful piece of information when deciding how to distribute the execution.

The following theorem states the soundness of Def. 7.3 when rules are transformed using Def. 7.2.

THEOREM 7.5. Let P be a program with an entry procedure p , and let $\hat{p}^{\text{ub}}(\bar{x})$ be a closed-form UB function for $\hat{p}(\bar{x}) \in \hat{S}_P$ where

\hat{S}_P is the cost relation generated after applying the `finish-async` transformation of Def. 7.2. Then, for any trace $t \equiv (\mathbf{A}_0 ; \langle 1, p(\bar{x}), tv \rangle) \rightarrow^* (\mathbf{A}_n ; T_n)$ it holds that $\hat{p}^{ub}(\bar{v}) + 1 \geq \text{peakAvailable}(t)$, where $\bar{v} = tv(\bar{x})$.

8. Improving with Surviving Tasks

In this section, our goal is to improve the accuracy of the UBs we have obtained in the previous sections by beholding which tasks *survive* after the method returns. The number of surviving tasks from a (normal) method call $q(\bar{x})$ refers to the number of tasks created during the execution of q which are alive after its local termination. Such surviving tasks could start their execution even after the local termination of $q(\bar{x})$. For an asynchronous call `async`{ $p(\bar{x})$ }, the number of tasks that can survive after p returns, in principle, bounded by its peak, and, for `finish`{ $q(\bar{x})$ }, is 0 by definition. In this section, we use this information in order to improve the peak of live and available tasks. We use the term “peak of tasks” to refer to any of the two (alive or available).

EXAMPLE 8.1. Consider the following program:

```
m ← p, Ⓢasync{q}
p ← async{q}, finish{h}, async{q}
h ← async{q}, async{q}, async{q}
```

and assume that q does not contain asynchronous calls. By Def. 7.3, we generate the following equations for the peak of available tasks:

$$\begin{aligned}\hat{m} &= \hat{p} + 1 + \hat{q} \\ \hat{p} &= 1 + \hat{q} + \max(2 + \hat{q} + \hat{q} + \hat{q}, 1 + \hat{q})\end{aligned}$$

which, since $\hat{q} = 0$, are solved to $\hat{m} = 4$. Let us explain how we can refine this peak using surviving information. While the peak of available tasks when executing p is 3, only 2 tasks can survive after p returns, i.e., they can be available after program point **Ⓢ**. The idea is that the peak of available tasks for m (ignoring the task in which m is being executed) can be defined as the maximum of the following two scenarios: (a) the peak of the tasks while executing p or (b) those that survive after p returns plus 1 for the last asynchronous call in m . This will lead to 3, which improves the previous peak by 1.

DEFINITION 8.2 (surviving tasks). Consider a program P with an entry procedure p , and a trace $t = (\mathbf{A} ; \langle 1, p(\bar{x}), tv \rangle) \rightarrow^* (\mathbf{A}_n ; T_n)$ such that p locally terminates before reaching T_n . The number of surviving tasks from p in t can be defined as $\text{surviving}(p) = |\text{available}(T_n)|$.

The following definition presents a novel form of equations, called *combined peak/surviving equations*, which take advantage of static knowledge on the surviving tasks in order to approximate the peak of tasks more accurately. Given a procedure $p(\bar{x})$, the main idea is to set up two kinds of relations: (1) the *peak equations* $\hat{p}(\bar{x})$, which define the peak of tasks reached during the execution of p ; and (2) the *surviving equations* $\check{p}(\bar{x})$, which define the surviving tasks from a call to $p(\bar{x})$. The definitions are mutually recursive.

DEFINITION 8.3 (combined peak/surviving equations). Let r be a rule, and φ_r be its corresponding value abstraction as in Def. 6.2. The combined peak/surviving equations for r consist of its surviving equation $\check{p}(\bar{x}) = \sum_{i=1}^n \mathcal{E}(b_i), \varphi_r$, where:

$$\begin{aligned}\mathcal{E}(b) &= 1 + \hat{q}(\bar{z}) && \text{if } b = \text{async}\{q(\bar{z})\} \\ \mathcal{E}(b) &= \check{q}(\bar{z}) && \text{if } b = q(\bar{z}) \\ \mathcal{E}(b) &= 0 && \text{otherwise}\end{aligned}$$

and its peak equation which is like in Def. 7.3, but letting $\mathcal{P}(b \cdot instr)$ be $\max(\hat{q}(\bar{z}), \check{q}(\bar{z}) + \mathcal{P}(instr))$ when $b = q(\bar{z})$.

Observe that this equation modifies Def. 6.2 in the case of a synchronous call in order to take advantage of the surviving information. In the surviving equation, we distinguish three cases: (i)

in asynchronous calls, the new task can survive, as well as any (the peak) tasks created by the callee; (ii) in synchronous calls, tasks escaping from such call are counted; (iii) the other instructions are mapped to 0 (e.g., we are sure that nothing survives after `finish`{ s }).

EXAMPLE 8.4. The solution of the following combined equations, obtained by applying Def. 8.3 to the rules of Ex. 8.1, corresponds to the improved peak UB, as explained in Ex. 8.1:

$$\begin{aligned}\hat{m} &= \max(\hat{p}, \check{p} + 1 + \hat{q}) && \check{m} = \check{p} + 1 + \hat{q} \\ \hat{p} &= 1 + \hat{q} + \max(2 + \hat{q} + \hat{q} + \hat{q}, 1 + \hat{q}) && \check{p} = 1 + \hat{q} + 1 + \hat{q}\end{aligned}$$

In the above example, the accuracy gain is constant. In general, it can be much larger (even in complexity order). Consider the program in Sec. 2.3 whose intermediate representation is:

$$\begin{aligned}f(n, i) &\leftarrow n \leq 0 \\ f(n, i) &\leftarrow n > 0, \text{finish-async}\{activity_a(i), activity_b(i)\}, \\ & n' := n - 1, i' := 2 * i + 1, i'' := 2i + 2, \\ & f(n', i'), f(n', i'')\end{aligned}$$

By applying Def. 8.3, we obtain the equations:

$$\begin{aligned}\check{f}(n, i) &= 0 && \{n \leq 0\} \\ \check{f}(n, i) &= \check{f}(n', i') + \check{f}(n', i'') && \varphi \\ \check{f}(n, i) &= 0 && \{n \leq 0\} \\ \check{f}(n, i) &= \max(1 + activity_a(i) + activity_b(i), \\ & \max(\hat{f}(n', i'), \check{f}(n', i') + \max(\hat{f}(n', i''), \check{f}(n', i'')))) && \varphi\end{aligned}$$

where the condition φ is $\{n > 0, n' = n - 1, i' = 2i + 1, i'' = 2i + 2\}$, and $activity_a(i) = activity_b(i) = 0$. Since $\check{f}(n, i)$ is solved to 0, the solution to the combined equations is the constant 1. Note that, applying Def. 5.4, we obtain the exponential bound shown in Sec. 2.3. Applying either Def. 6.2 or Def. 7.3, we obtain an exponential bound as well. Hence, the solution of the combined equations is much more accurate than all previous solutions.

Soundness states that \hat{p} and \check{p} correctly approximate the peak of available tasks and the surviving tasks, respectively.

THEOREM 8.5. Let P have an entry procedure p , and q be defined in P . Let $\hat{p}^{ub}(\bar{x})$ be a closed-form UB function for its combined peak/surviving equations. Given a trace $t \equiv (\mathbf{A}_0 ; \langle 1, p(\bar{x}), tv \rangle) \rightarrow^* (\mathbf{A}_n ; T_n)$. Then, letting $\bar{v} = tv(\bar{x})$, it holds that (1) $\hat{p}^{ub}(\bar{v}) + 1 \geq \text{peakAvailable}(t)$; and (2) $\check{p}^{ub}(\bar{v}) + 1 \geq \text{surviving}(t)$.

Note that, if we use the peak equations as in Def. 6.2, instead of point (1) above, it holds that $\hat{p}^{ub}(\bar{v}) + 1 \geq \text{peakLive}(t)$.

We note that this improvement might come at the price of efficiency and effectiveness of the analysis. As regards efficiency, the fact that for each procedure in the program, we generate two sets of equations increases the analysis time. In particular, the time required to infer closed-form UBs for the combined relations almost doubles. As regards effectiveness, the fact that the definition of both relations is mutually recursive can make their solving more complex. Nonetheless, the mutual recursion disappears in many cases (e.g., when the number of surviving tasks is constant). Also, certain solvers can solve such kind of recursion. After solving the equations, the obtained UBs are guaranteed to be strictly more precise than those obtained in the previous sections.

9. Experimental Results

We have implemented our technique within the COSTA System, a COST and Termination Analyzer for Java bytecode which can be tried out online at <http://costa.ls.fi.upm.es/costa> by choosing the example folder “x10” in the analyzer. The experimental evaluation has been performed on a set of small but representative X10 programs (available at the X10 website <http://>

#	$U_T/U_A/U_E$	ms	#	$U_T/U_A/U_E$	ms
1	$(N-1)(\log N)$	500	2	$61441N+61441$	310
	$(N-1)(\log N)$	310		$61441N+61441$	270
	$N-1$	450		61569	460
3	$2048N+48$	240	4	$2^{N-1}-1$	200
	$2048N+48$	260		$2^{N-1}-1$	210
	$1024N+16$	390		$2^{N-1}-1$	240
5	kN^3+3kN^2+kN	830	6	$50 * (2N+2000)$	170
	$(k+1)N^3+(k+2)N^2+(k+1)N$	760		$\max(N, 2000)$	170
	$(k+1)N^3+(2k+3)N^2+(k+3)N+1$	1340		$\max(N, 2000)$	210
7	$10N_1N_2$	2680	8	N	100
	$10N_1N_2$	1780		1	90
	$N_1N_2 + N_1$	2850		1	140

Table 1. Benchmarks: 1 ArraySum (1044 Kb); 2 CUDABlackScholes (1071); 3 FRASimpleDist (1134); 4 Fib (717); 5 HeatTransfer_v1 (1913); 6 KMeansDist (1124); 7 PLU_2.C (8520); 8 method print()V of SparseMat (706).

x10-lang.org/) containing interesting parallelism patterns. The implementation includes existing tools developed for Java which translate the original program into the IR. Concretely, the examples have been first (manually) translated from X10 to Java, preserving the structure of the parallelism. From this point on, the analysis is fully automatic. In some cases, purely numerical computations have been omitted (e.g., most of the method doBlackScholes in CUDABlackScholes), and pieces of code which manipulate data structures in a way that is specific to X10 have been simplified. *Places* have been ignored. Also, to avoid virtual invocations that often complicates the analysis, we sometimes translate calls $o.m()$ to $m(o)$, and define m as a static method. Finally, `async` and `finish` statements have been simulated (only for the sake of the analysis, not for actual execution in the JVM) by means of special method calls. Overall, the translation is done in such a way that the Java code arguably preserves the properties of interest.

The results are shown in Table 1. For each benchmark, the total number U_T of spawned tasks (first row), the peak U_A of live tasks (second row), and the refined peak U_E of live tasks using surviving information (third row) are inferred. We do not add 1 for the initial task. Most examples take as input a numerical parameter, which is a measure of the size of the problem. Such parameter is usually taken to be the length of the array of String which is the argument of the main method, and appears as N in the table (N_1 and N_2 if the input consists of two parameters). In two cases, U_A is better than U_T , meaning that the analysis was able to infer that some tasks cannot be alive at the same time. Moreover, U_E improves on U_A in four examples, thus showing the usefulness of considering surviving information. The table also shows (next to the name of the benchmark) the size in Kbytes of the (transformed) .class file, and the total analysis time ms in milliseconds.

Let us explain the results in more detail. ArraySum is interesting because the sum is executed many times under different assumptions about the number of tasks which are going to be spawned: at each iteration, this number is multiplied by 2 (starting from 1) until a threshold N is reached (note that the X10 code uses a constant threshold 4, so that our version is, in some sense, more general). The result is that at most $N-1$ tasks are spawned at each one of the $\log N$ iterations, thus giving a total of $(\log N) * (N-1)$ tasks. On the other hand, due to the finish statement which wraps each iteration, only $N-1$ tasks can be alive at the same time, thus giv-

ing such number as U_E . Note that the analysis of live tasks needs surviving information in order to get the linear upper bound.

In CUDABlackScholes, N is the number of iterations which is the constant 512 in the original program. It can be seen that U_T is bigger since every iteration is performed inside a finish statement, so that tasks created in different iterations cannot be alive at the same time. The UB of Fib is exponential due to the structure of the recursive calls. The total number and the peak number of tasks are equal; indeed, all spawned tasks can be alive at the same time.

In HeatTransfer_v1, the UB is cubic in all cases, since the operations on the data structures spawn a cubic number of tasks, and all tasks are alive at the same time since a single finish statement wraps this part of the code. The difference (not in the order of magnitude) between the UBs is due to the different loss of precision when solving the equations. The number of iterations of the loop in run() depends on the guard $\delta < \epsilon$ on double numbers. This bound is unpredictable by most state-of-the-art static analyzers, so that the program has been modified in order to iterate a fixed number of times k . In KMeansDist, the constants 2000 and 50 appearing in the UBs are constants in the X10 code, while N is a measure of the size of the data structure. In the biggest example PLU_2.C, considering surviving information allows to remove a constant factor 10 which is a constant in the program code.

Overall, we argue that, although our implementation is still a prototype, the experiments show that our approach is promising and leads (fully automatically) to reasonably accurate task-level UBs.

10. Related Work

As regards the language, several subsets of X10 [1, 16, 20] have been defined in the literature. For the parallel part of the language, the subset we consider is like FX10 [16], while the sequential part is richer since not handling recursion would have been an important restriction for the task-level analysis. The majority of related work around the X10 language is on may-happen-in-parallel analysis [16] and determinism [26]. This is a complementary line of research to ours, in the sense that we can use the results of such analyses to improve ours, as we will discuss in Sec. 11.

Due to our interpretation of the task level of a program as a resource consumed along its execution, our work is more directly related to cost-analysis (or resource-usage-analysis) frameworks [3, 5, 12, 13, 24]. All such frameworks assume a sequential execution model. Moreover, they often are applied to measure accumulative resources. Another non-accumulative resource is memory consumption in the presence of garbage collection. There has been a respectable development in heap space analysis for Java-like and functional languages [5, 7, 9, 14, 24] during the last years. Among them, our work is more related to those that rely on RR [5, 24]. Still, heap space bounds are fundamentally different from task-level bounds as, in the case of memory, the challenge is to model the behavior of the garbage collector at the level of the cost equations. In our case, the challenge is to handle concurrency and to capture in the equations the states in which tasks terminate.

11. Conclusions and Future Work

We have presented a novel static analysis to approximate the task level of parallel X10-like programs. Our approach is based by the view that the task level of a program is a particular (non-accumulative) resource consumed along its (parallel) execution. Existing cost-analysis frameworks assume a standard *sequential* programming model on resources which are typically *accumulative*. It is clear that both these deviations from existing frameworks add significant complexity to the problem of inferring task-level bounds. Our key contribution is the generation of task-level *recurrence relations* that soundly and accurately approximate the task level of the program in the parallel setting. An important obser-

vation (and a side-effect contribution of our work, and due to the characteristics of X10) is that obtaining an UB from the RR implies bounding the number of iterations of loops in the original X10 program. Therefore, our work indirectly provides a *global termination analysis* for X10 programs. In other words, if the analysis finds a task-level UB, it is guaranteed that the original X10 program terminates for any input data.

The abstraction performed by the value analysis component, though simple, ensures that the UBs obtained are sound for any particular task scheduler. One direction for future work is to improve the precision of the analysis by enriching the value analysis assuming a particular scheduling. To do this, we first need to make some assumption on the policy which establishes which tasks run in parallel. Then, we can reuse existing may-happen-in-parallel analyses [16] which specifically treat the `async-finish` constructs of X10. The output of such analysis annotates each instruction with the set of instructions that can be executed in parallel with it. One could then prove that the fragments of code which might be executed in parallel are independent [26] (i.e., they do not read/write on the same global data). In such case, we can then use existing field-sensitive value analyses [17] developed for similar languages in order to improve the precision of our UBs.

As another direction for future work, we plan to extend our analysis to the full X10 language. In particular, we believe that handling places can give us some interesting results. This requires enhancing the `async` construct as `async{s, id}`, with the identifier `id` of the server encharged of running the task `s` asynchronously. An interesting application of our analysis in this setting is to infer the throughput of the different servers of the system, which could be very useful to balance the workload in distributed applications.

Our approach can be easily adapted to count the peak at a program point, i.e., the maximum number of tasks that can be alive (or available) in parallel at that specific program point. Suppose that the program point of interest is `@`, then we can modify Def. 6.2 as follows: we add $\mathcal{P}(\text{@} \cdot \text{instr}) = 1 + \mathcal{P}(\text{instr})$, and remove the constant 1 from the equation of `async`. Such information is useful, for example, when, at the program point of interest, we query a server: the obtained UB indicates the load of the server.

Acknowledgments

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

- [1] M. Abadi and G. D. Plotkin. A model of cooperative threads. In *Proc. of POPL'09*, pages 29–40. ACM, 2009.
- [2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
- [3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
- [4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Proc. of FMCO'07*, volume 5382 of *LNCS*, pages 113–133. Springer, 2008.
- [5] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric Inference of Memory Requirements for Garbage Collected Languages. In *Proc. of ISMM'10*, pages 121–130. ACM Press, 2010.
- [6] A. W. Appel. Ssa is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.
- [7] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proc. of ISMM'08*, pages 141–150. ACM Press, 2008.
- [8] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster computing. In *Proc. of OOP-SLA'05*, pages 519–538. ACM, 2005.
- [9] W.-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proc. of ISMM'08*, pages 151–160. ACM Press, 2008.
- [10] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
- [11] M. Fähndrich. Static Verification for Code Contracts. In *Proc. of SAS'10*, volume 6337 of *LNCS*, pages 2–5. Springer, 2010.
- [12] 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.
- [13] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Proc. of ESOP'10*, volume 6012 of *LNCS*, pages 287–306. Springer, 2010.
- [14] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *Proc. of ESOP'06*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006.
- [15] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
- [16] J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Proc. of PPoPP'10*, pages 25–36. ACM, 2010.
- [17] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *Proc. of LCTES'06*, pages 54–63. ACM, 2006.
- [18] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proc. of RTA'10*, volume 6 of *LIPICs*, pages 259–276, 2010.
- [19] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proc. of RV'10*, volume 6418 of *LNCS*, pages 368–383. Springer, 2010.
- [20] V. A. Saraswat and R. Jagadeesan. Concurrent Clustered Programming. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*, pages 353–367. Springer, 2005.
- [21] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode based on Path-Length. *Transactions on Programming Languages and Systems*, 32(3), 2010.
- [22] W. Zou T. Wei, J. Mao and Y. Chen. A new algorithm for identifying loops in decompilation. In *Proc. of SAS'07*, volume 4634 of *LNCS*, pages 170–183, 2007.
- [23] L. Unnikrishnan and S. Stoller. Parametric heap usage analysis for functional programs. In *Proc. of ISMM'09*, pages 139–148. ACM Press, 2009.
- [24] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85. Springer, 2003.
- [25] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *Proc. of CASCON'99*, pages 125–135. IBM, 1999.
- [26] M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic Verification of Determinism for Structured Parallel Programs. In *Proc. of SAS'10*, volume 6337 of *LNCS*, pages 455–471. Springer, 2010.
- [27] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.