May-Happen-in-Parallel Analysis for Priority-based Scheduling (Extended Version)

Technical Report SIC-12-13*
Dpto. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Revised October 23, 2013

Elvira Albert, Samir Genaim, and Enrique Martin-Martin

Complutense University of Madrid, Spain

Abstract. A *may-happen-in-parallel* (MHP) analysis infers the sets of pairs of program points that may execute in parallel along a program's execution. This is an essential piece of information to detect data races, and also to infer more complex properties of concurrent programs, e.g., deadlock freeness, termination and resource consumption analyses can greatly benefit from the MHP relations to increase their accuracy. Previous MHP analyses have assumed a worst case scenario by adopting a simplistic (non-deterministic) task scheduler which can select any available task. While the results of the analysis for a non-deterministic scheduler are obviously sound, they can lead to an overly pessimistic result. We present an MHP analysis for an asynchronous language with *prioritized* tasks buffers. *Priority-based scheduling* is arguably the most common scheduling strategy adopted in the implementation of concurrent languages. The challenge is to be able to take task priorities into account at static analysis time in order to filter out unfeasible MHP pairs.

1 Introduction

In asynchronous programming, programmers divide computations into shorter tasks which may create additional tasks to be executed asynchronously. Each task is placed into a task-buffer which can execute in *parallel* with other task-buffers. The use of a *synchronization* mechanism enables that the execution of a task is synchronized with the completion of another task. Synchronization can be performed via shared-memory [9] or via future variables [13, 8]. Concurrent *interleavings* in a buffer can occur if, while a task is awaiting for the completion of another task, the processor is released such that another pending task can start

* Extended version of the paper *Termination and Cost Analysis of Loops with Concurrent Interleavings* that will appear in the *19th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning* (LPAR-19) and published by Springer in a *LNCS ARCoSS* volume.

to execute. This programming model captures the essence of the concurrency models in X10 [13], ABS [12], Erlang [1] and Scala [11], and it is the basis of actor-like concurrency [2, 11]. The most common strategy to schedule tasks is undoubtedly *priority-based scheduling*. Each task has a priority level such that when the active task executing in the buffer releases the processor, a highest priority pending task is taken from its buffer and begins executing. Asynchronous programming with *prioritized tasks buffers* has been used to model real-world asynchronous software, e.g., Windows drivers, engines of modern web browsers, Linux's work queues, among others (see [9] and its references).

The higher level of abstraction that asynchronous programming provides, when compared to lower-level mechanisms like the use of multi-threading and locks, allows writing software which is more reliable and more amenable to be analyzed. In spite of this, proving error-freeness of these programs is still quite challenging. The difficulties are mostly related to: (1) *Tasks interleavings*, typically a programmer decomposes a task t into subtasks $t_1, \ldots, t_n$. Even if each of the sub-tasks would execute serially, it can happen that a task k unrelated to this computation interleaves its execution between t_i and t_{i+1}. If this task k changes the shared-memory, it can interfere with the computation in several ways, e.g., leading to non-termination, to an unbounded resource consumption, and to deadlocks. (2) *Buffers parallelism*, tasks executing across several task-buffers can run in parallel, this could lead to deadlocks and data races.

In this paper, we present a *may-happen-in-parallel* (MHP) analysis which identifies pairs of statements that can execute in parallel and in an interleaved way (see [13, 3]). MHP is a crucial analysis to later prove the properties mentioned above. It directly allows ensuring absence of data races. Besides, MHP pairs allow us to greatly improve the accuracy of deadlock analysis [16, 10] as it discards unfeasible deadlocks when the instructions involved in a possible deadlock cycle cannot happen in parallel. Also, it improves the accuracy of termination and cost analysis [5] since it allows discarding unfeasible interleavings. For instance, consider a loop like `while (l!=null) {x=b.m(l.data); await x?; l=l.next;}`, where `x=b.m(e)` posts an asynchronous task `m(e)` on buffer `b`, and the instruction `await x?` synchronizes with the completion of the asynchronous task by means of the future variable `x`. If the asynchronous task is not completed (`x` is not ready), the current task releases the processor and another task can take it. This loop terminates provided no instruction that increases the length of the list `l` *interleaves* or *executes in parallel* with the body of this loop.

Existing MHP analyses [13, 3] assume a worst case scenario by adopting a simplistic (non-deterministic) task scheduler which can select any available task. While the results of the analysis for a non-deterministic scheduler are obviously sound, they can lead to an overly pessimistic result and report false errors due to unfeasible schedulings in the task order selection. For instance, consider two buffers `b1` and `b2` and assume we are executing a task in `b1` with the following code "`x=b1.m1(e1); y=b1.m2(e2); await x?; b2.m3(e3);`". If the priority of the task executing `m1` is smaller than that of `m2`, then it is ensured that task `m2` and `m3` will not execute in parallel even if the synchronization via **await** is

on the completion of `m1`. This is because at the **await** instruction, when the processor is released, `m2` will be selected by the priority-based scheduler before `m1`. A non-deterministic scheduler would give this spurious parallelism.

Our starting point is the MHP analysis for non-deterministic scheduling of [3], which distinguishes a local phase in which one inspects the code of each task locally, and ignores transitive calls, and a global phase in which the results of the local analysis are composed to build a global *MHP-graph* which captures the parallelism with transitive calls and among multiple task-buffers. The contribution of this paper is an MHP analysis for a priority-based scheduling which takes priorities into account both at the local and global levels of the analysis. As each buffer has its own scheduler which is independent of other buffer's schedulers, priorities can be only applied to establish the order of execution among the tasks executing on the same task-buffer (*intra-buffer* MHP pairs). Interestingly, even by only using priorities at the intra-buffer level, we are also able to implicitly eliminate unfeasible *inter-buffer* MHP pairs. We have implemented our analysis in the MayPar system [4] and evaluated it on some challenging examples, including some of the benchmarks used in [9]. The system can be used online through a web interface where the benchmarks used are also available.

2 Language

We consider asynchronous programs with priority-levels and multiple tasks buffers. Tasks can be synchronized with the completion of other tasks (of the same or of a different buffer) using futures. In this model, only highest-priority tasks may be dispatched, and tasks from different task buffers execute in parallel. The number of task buffers does not have to be known a priori and task buffers can be dynamically created. We keep the concept of task-buffer disconnected from physical entities, such as processes, threads, objects, processors, cores, etc. In [9], particular mappings of task-buffers to such entities in real-world asynchronous systems are described. Our model captures the essence of the concurrency and distribution models used in X10 [13] and in actor-languages (including ABS [12], Erlang [1] and Scala [11]). It also has many similarities with [9], the main difference being that the synchronization mechanism is by means of future variables (instead of using the shared-memory for this purpose).

2.1 Syntax

Each program declares a sequence of global variables $g_0, \ldots, g_n$ and a sequence of methods named $m_0, \ldots, m_i$ (that may declare local variables) such that one of the methods, named main, corresponds to the initial method which is never posted or called and it is executing in a buffer with identifier `0`. The grammar below describes the syntax of our programs. Here, T are types, m procedure names, e expressions, x can be global or local variables, buffer identifiers b are local variables, f are future variables, and priority levels p are natural numbers.

$$
\begin{array}{lll}
M & ::= & T\ m(\bar{T}\ \bar{x})\{s; \textbf{return}\ e;\} \\
s & ::= & s; s \mid x = e \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \textbf{do}\ s \mid \\
 & & \textbf{await}\ f? \mid b = \mathsf{newBuffer} \mid f = b.m(\langle \bar{e} \rangle, p) \mid \textbf{release}
\end{array}
$$

The notation $\bar{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. We use the special buffer identifier *this* to denote the current buffer. For the sake of generality, the syntax of expressions is left free and also the set of types is not specified. We assume that every method ends with a **return** instruction.

The concurrency model is as follows. Each buffer has a lock that is shared by all tasks that belong to the buffer. Data synchronization is by means of future variables as follows. An **await** y? instruction is used to synchronize with the result of executing task $y{=}b.m(\langle \bar{z} \rangle, p)$ such that **await** y? is executed only when the future variable y is available (and hence the task executing m is finished). In the meantime, the buffer's lock can be released and some highest priority *pending* task on that buffer can take it. The instruction **release** can be used to unconditionally release the processor so that other pending task can take it. Therefore, our concurrency model is *cooperative* as processor release points are explicit in the code, in contrast to a *preemptive* model in which a higher priority task can interrupt the execution of a lower priority task at any point (see Sec. 7). W.l.o.g, we assume that all methods in a program have different names.

2.2 Semantics

A *program state* $St = \langle g, \texttt{Buf} \rangle$ is a mapping g from the global variables to their values along with all created buffers `Buf`. `Buf` is of the form $\mathit{buffer}_1 \parallel \ldots \parallel \mathit{buffer}_n$ denoting the parallel execution of the created task-buffers. Each *buffer* is a term $\mathit{buffer}(\mathit{bid}, \mathit{lk}, \mathcal{Q})$ where *bid* is the buffer identifier, *lk* is the identifier of the *active task* that holds the buffer's lock or $\perp$ if the buffer's lock is free, and $\mathcal{Q}$ is the set of tasks in the buffer. Only one task can be *active* (running) in each buffer and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $\mathit{tsk}(\mathit{tid}, m, p, l, s)$ where *tid* is a unique task identifier, m is the method name executing in the task, p is the task priority level (the larger the number, the higher the priority), l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created buffers and tasks never disappear from the state.

The execution of a program starts from an initial state where we have an initial buffer with identifier 0 executing task 0 of the form $S_0 = \langle g, \mathit{buffer}(0, 0, \{\mathit{tsk}(0, \mathsf{main}, p, l, \mathit{body}(\mathsf{main}))\}) \rangle$. Here, g contains initial values for the global variables, l maps parameters to their initial values and local reference and future variables to **null** (standard initialization), p is the priority given to main, and $\mathit{body}(m)$ refers to the sequence of instructions in the method m. The execution proceeds from S_0 by selecting *non-deterministically* one of the buffers and applying the semantic rules depicted in Fig. 1. We omit the treatment of the sequential instructions as it is standard, and we also omit the global memory g from the state as it is only modified by the sequential instructions.

$$\text{(NEWBUFFER)}\ \frac{\text{fresh}(bid')\ ,\ l' = l[x \to bid'],\ t = tsk(tid, m, p, l, \langle x = \mathsf{newBuffer}; s\rangle)}{\begin{array}{c}buffer(bid, tid, \{t\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, tid, \{tsk(tid, m, p, l', s)\} \cup \mathcal{Q}) \parallel buffer(bid', \bot, \{\}) \parallel B\end{array}}$$

$$\text{(PRIORITY)}\ \frac{highestP(\mathcal{Q}) = tid,\ t = tsk(tid, \_, \_, \_, s) \in \mathcal{Q},\ s \neq \epsilon(v)}{buffer(bid, \bot, \mathcal{Q}) \parallel B \rightsquigarrow buffer(bid, tid, \mathcal{Q}) \parallel B}$$

$$\text{(ASYNC)}\ \frac{l(x) = bid_1,\ \text{fresh}(tid_1),\ l' = l[y \to tid_1],\ l_1 = buildLocals(\bar{z}, m_1)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle y = x.m_1(\bar{z}, p_1); s\rangle\} \cup \mathcal{Q}) \parallel buffer(bid_1, \_, \mathcal{Q}') \parallel B \rightsquigarrow \\ buffer(bid, tid, \{tsk(tid, m, p, l', s)\} \cup \mathcal{Q}) \parallel \\ buffer(bid_1, \_, \{tsk(tid_1, m_1, p_1, l_1, body(m_1))\} \cup \mathcal{Q}') \parallel B\end{array}}$$

$$\text{(AWAIT1)}\ \frac{l(y) = tid_1,\ tsk(tid_1, \_, \_, \_, s_1) \in \mathtt{Buf}, s_1 = \epsilon(v)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, tid, \{tsk(tid, m, p, l, s)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$\text{(AWAIT2)}\ \frac{l(y) = tid_1,\ tsk(tid_1, \_, \_, \_, s_1) \in \mathtt{Buf}, s_1 \neq \epsilon(v)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, \bot, \{tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$\text{(RELEASE)}\ \frac{}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{release}; s\rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, \bot, \{tsk(tid, m, p, l, s)\} \cup \mathcal{Q}) \parallel B\end{array}}$$

$$\text{(RETURN)}\ \frac{v = l(x)}{\begin{array}{c}buffer(bid, tid, \{tsk(tid, m, p, l, \langle \mathsf{return}\ x; \rangle)\} \cup \mathcal{Q}) \parallel B \rightsquigarrow \\ buffer(bid, \bot, \{tsk(tid, m, p, l, \epsilon(v))\} \cup \mathcal{Q}) \parallel B\end{array}}$$

Fig. 1. Summarized Semantics for a Priority-based Scheduling Async Language

NEWBUFFER: an active task *tid* in buffer *bid* creates a buffer bid' which is introduced to the state with a free lock. PRIORITY: Function *highestP* returns a highest-priority task that is not finished, and it obtains its buffer's lock. ASYNC: A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 which is associated to the corresponding future variable y in l'. We have assumed that $bid \neq bid_1$, but the case $bid = bid_1$ is analogous, the new task tid_1 is simply added to $\mathcal{Q}$ of *bid*. AWAIT1: If the future variable we are awaiting for points to a finished task, the **await** can be completed. The finished task t_1 is looked up in all buffers in the current state (denoted `Buf`). AWAIT2: Otherwise, the task yields the lock so that any other task of the same buffer can take it. RELEASE: the current task frees the lock. RETURN: When **return**

```
1  // g1 global variable
2  // g2 global variable
3  void task(){
4      g2 = g2 + 1;
5  }
6  void f(){
7      while( g1 > 0 ){
8        g1 = g1 − 1;
9        g2 = g2 + 1;
10       release;
11     }
12 }
13 void m(){
14     while( g1 < 0 ){
15       g1 = g1 + 1;
16       release;
17     }
18 }
19 void h(){
20     while(g1 > 0){
21       g1 = g1 − 2;
22       release;
23     }
24 }
25 // main has priority 0
26 main(){
27     this.f(<>,10);
28     Fut x = this.m(<>,5);
29     await x?;
30     this.h(<>,10);
31     Buffer o=newbuffer;
32     o.task(<>,0);
33     ...
34 }
```

Fig. 2. Example for inter-buffer and intra-buffer may-happen-in-parallel relations

is executed, the return value is stored in v so that it can be obtained by the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$) but it does not disappear from the state as its return value may be needed later on in an **await**.

Example 1. Figure 2 shows some simple methods which will illustrate different aspects of our analysis. In particular, non-termination of certain tasks and data races can occur if priorities are not properly assigned by the programmer, and later considered by the analysis. Our analysis will take the assigned priorities into account in order to gather the necessary MHP information to be able to guarantee termination and absence of data races. Let us by now only show some execution steps. The execution starts from a buffer 0 with a single task in which we are executing the main method. Let us assume that such task has been given the lowest priority 0. The global memory g is assumed to be properly initialized.

$St_0 \equiv \langle g, \mathit{buffer}(0, 0, \{tsk(0, \mathsf{main}, 0, l, \mathit{body}(\mathsf{main}))\})\rangle \xrightarrow{async}$

$St_1 \equiv \langle g, \mathit{buffer}(0, 0, \{tsk(0, ..), tsk(1, f, 10, ..)\})\rangle \xrightarrow{async}$

$St_2 \equiv \langle g, \mathit{buffer}(0, 0, \{tsk(0, ..), tsk(1, ..), tsk(2, m, 5..)\})\rangle \xrightarrow{await}$

$St_3 \equiv \langle g, \mathit{buffer}(0, \bot, \{tsk(0, .., await), tsk(1, ..), tsk(2, m, 5..)\})\rangle \xrightarrow{priority}$

$St_4 \equiv \langle g, \mathit{buffer}(0, 1, \{tsk(0, .., await), tsk(1, ..), tsk(2, m, 5..)\})\rangle \rightarrow^*$

$St_5 \equiv \langle g', \mathit{buffer}(0, 1, \{tsk(0, .., await), tsk(1, .., return), tsk(2, m, 5..)\})\rangle \xrightarrow{return}$

$St_6 \equiv \langle g', \mathit{buffer}(0, \bot, \{tsk(0, .., await), tsk(1, .., \epsilon(v)), tsk(2, m, 5..)\})\rangle \xrightarrow{priority}$

$St_7 \equiv \langle g', \mathit{buffer}(0, 2, \{tsk(0, .., await), tsk(1, .., \epsilon(v)), tsk(2, m, 5..)\})\rangle \rightarrow^*$

$St_8 \equiv \langle g'', \mathit{buffer}(0, 0, \{tsk(0, ..), tsk(1, .., \epsilon(v)), tsk(2, .., \epsilon(v)), tsk(3..)\})\rangle \xrightarrow{newbuf}$

$St_9 \equiv \langle g'', \mathit{buffer}(0, 0, \{tsk(0..), tsk(1..), tsk(2..), tsk(3..)\}), \rangle \mathit{buffer}(1, \bot, \{\}) \xrightarrow{async}$

$St_{10} \equiv \langle g'', \mathit{buffer}(0, 0, \{tsk(0..), ..\}), \rangle \mathit{buffer}(1, \bot, \{task(4..)\}) \xrightarrow{priority}$

$St_{11} \equiv \langle g'', \mathit{buffer}(0, 0, \{tsk(0..), ..\}), \rangle \mathit{buffer}(1, 4, \{task(4..)\}) \xrightarrow{\cdots}$

At St_1, we execute the instruction at Line 27 (L27 for short) that posts, in the current buffer **this**, a new task (with identifier 1) that will execute method f with priority 10. The next step St_2 posts another task (with identifier 2) in the current buffer with a lower priority (namely 5). At St_3, an **await** instruction (L29) is used to synchronize the execution with the completion of the task 2 spawned at L28. As the task executing f has higher priority than the one executing m, it will be selected for execution at St_4. After returning from the execution of task 1 in St_5, the PRIORITY rule selects task 2 for execution in St_6. An interesting aspect is that after creating buffer 1 at St_{10}, execution can non-deterministically choose buffer 0 or 1 (in St_{11} buffer 1 has been selected).

3 Definition of MHP

We first formally define the concrete property "MHP" that we want to approximate using static analysis. In what follows, we assume that instructions are labelled such that it is possible to obtain the corresponding program point identifiers. We also assume that program points are globally different. We use $p_{\mathring{m}}$ to refer to the entry program point of method m, and $p_{\dot{m}}$ to all program points after its **return** instruction. The set of all program points of P is denoted by $P_{\mathcal{P}}$. We write $p \in m$ to indicate that program point p belongs to method m. Given a sequence of instructions s, we use $pp(s)$ to refer to the program point identifier associated with its first instruction and $pp(\epsilon(v)) = p_{\dot{m}}$.

Definition 1 (concrete MHP). *Given a program P, its MHP is defined as $\mathcal{E}_P = \cup\{\mathcal{E}_S | S_0 \rightsquigarrow^* S\}$ where for $S = \langle g, \texttt{Buf}\rangle$, the set $\mathcal{E}_S$ is $\mathcal{E}_S = \{(pp(s_1), pp(s_2)) \mid buffer(bid_1, \_, \mathcal{Q}_1) \in \texttt{Buf}, buffer(bid_2, \_, \mathcal{Q}_2) \in \texttt{Buf}, t_1 = tsk(tid_1, \_, \_, \_, s_1) \in \mathcal{Q}_1, t_2 = tsk(tid_2, \_, \_, \_, s_2) \in \mathcal{Q}_2, tid_1 \neq tid_2\}$.*

The above definition considers the union of the pairs obtained from all derivations from S_0. This is because execution is non-deterministic in two dimensions: (1) in the selection of the buffer that is chosen for execution, since the buffers have access to the global memory different behaviours (and thus MHP pairs) can be obtained depending on the execution order, and (2) when there is more than one task with the highest priority, the selection is non-deterministic.

The MHP pairs can originate from *direct* or *indirect* task creation relationships. For instance, the parallelism between the points of the tasks executing h and task is *indirect* because they do not invoke one to the other directly, but a third task main invokes both of them. However, the parallelism between the points of the task main and those of task is *direct* because the first one invokes directly the latter one. Def. 1 captures all these forms of parallelism.

Importantly, $\mathcal{E}_P$ includes both *intra-buffer* and *inter-buffer* MHP pairs, each of which are relevant for different kinds of applications, as we explain below.

Intra-buffer MHP pairs. Intra-buffer relations in Def. 1 are pairs in which $bid_1 \equiv bid_2$. We always have that the first instructions of all tasks which are pending in the buffer's queue may-happen-in-parallel among them, and also with the instruction of the task which is currently active (has the buffer's lock). This piece of information allows approximating the tasks interleavings that we may have in a considered buffer. In particular, when the execution is at a processor release point, we use the MHP pairs to see the instructions that may execute if the processor is released. Information about task interleavings is essential to infer termination and resource consumption in any concurrent setting (see [5]).

Example 2. Consider the execution trace in Ex. 1, we have the MHP pairs (29,$p_{\hat{f}}$) and (29,$p_{\hat{m}}$) since when the active task 0 is executing the **await** (point 29) in St_4, we have that tasks 1 and 2 are pending at their entry points. The following execution steps give rise to many other MHP pairs. The most relevant point to note is that in St_8 when the execution is at L30 and onwards, the tasks 1 and 2 are guaranteed to be at their exit program points $p_{\hat{f}}$ and $p_{\hat{m}}$. Thus, we will not have any MHP pair between the instructions that update the global variable `g1` (L8 and L15 in tasks 1 and 2, resp.) and the release point at L22 of the task 3 executing `h`. This information is essential to prove the termination of `h`, as the analysis needs to be sure that the loop counter cannot be modified by instructions of other tasks that may execute in parallel with the body of this loop. The information is also needed to obtain an upper bound on the number of iterations of the loop and then infer the resource consumption of `h`.

Inter-buffer MHP pairs. In addition to intra-buffer MHP relations, *inter-buffer* MHP pairs happen when $bid_1 \neq bid_2$. In this case, we obtain the instructions that may execute in parallel in different buffers. This information is relevant at least for two purposes: (1) to detect data-races in the access to the global memory and (2) to detect deadlocks and livelocks when one buffer is awaiting for the completion of one task running in another buffer, while such other task is awaiting for the completion of the current task, and the execution of these (synchronization) instructions happens in parallel (or simultaneously). If the language allows blocking the execution of the buffer such that no other pending task can take it, we have a deadlock, otherwise we have a livelock.

Example 3. Consider again the execution trace in Ex. 1, in St_{10} we have created a new buffer 1 in which task 4 starts to execute at St_{11}. We will have the inter-buffer pair (21,4) as we can have L21 executing in buffer 0 and L4 executing in buffer 1. Note that, if `task` had updated `g1` instead of updating `g2`, we would have had a data race. Data races can lead to different types of errors, and static analyses that detect them are of utmost importance.

4 Method-Level Analysis with Priorities

In this section, we present the local phase of our MHP analysis which assigns to each program point, of a given method, an abstract state that describes the

$$
\begin{array}{lrll}
(1) & \tau_{\mathtt{p}}(y{=}\textsf{this}.m(\bar{x},p),M) & = M[\langle y,O,Z,R\rangle/\langle\star,O,Z,R\rangle]\cup\{\langle y,\mathtt{t},\check{m},p\rangle\} & \\
(2) & \tau_{\mathtt{p}}(y{=}x.m(\bar{x},p),M) & = M[\langle y,O,Z,R\rangle/\langle\star,O,Z,R\rangle]\cup\{\langle y,\mathtt{o},\tilde{m},p\rangle\} & \\
(3) & \tau_{\mathtt{p}}(\textsf{release},M) & = \tau_{\mathtt{p}}(\textsf{release}_1;\textsf{release}_2,M) & \\
(4) & \tau_{\mathtt{p}}(\textsf{release}_1,M) & = M[\langle Y,\mathtt{t},\check{m},p\rangle/\langle Y,\mathtt{t},\tilde{m},p\rangle] & \text{where } p\geq\mathtt{p} \\
(5) & \tau_{\mathtt{p}}(\textsf{release}_2,M) & = M[\langle Y,\mathtt{t},\check{m},p\rangle/\langle Y,\mathtt{t},\hat{m},p\rangle] & \text{where } p>\mathtt{p} \\
(6) & \tau_{\mathtt{p}}(\textsf{await } y?,M) & = M'[\langle y,O,\tilde{m},R\rangle/\langle y,O,\hat{m},R\rangle] & \\
 & & \text{where } M'=\tau_{\mathtt{p}}(\textsf{release}_1;\textsf{release}_2,M) & \\
(7) & \tau_{\mathtt{p}}(\textsf{return},M) & = M[\langle Y,\mathtt{t},\check{m},R\rangle/\langle Y,\mathtt{t},\tilde{m},R\rangle] & \\
(8) & \tau_{\mathtt{p}}(b,M) & = M \quad \text{otherwise} &
\end{array}
$$

Fig. 3. Method-level MHP transfer function: $\tau_{\mathtt{p}} : s \times \mathcal{B} \mapsto \mathcal{B}$.

status of the tasks that have been locally invoked so far. The status of a task can be (1) *pending*, i.e., it is at the entry program point; (2) *finished*, i.e., it has executed a **return** instruction already; or (3) *active*, i.e., it can be executing at any program point (including the entry and the exit). The analysis uses *MHP atoms* which are syntactic objects of the form $\langle F, O, T, R\rangle$ where

- F is either a valid future variable name or $\star$. The value $\star$ indicates that the task might not be associated with any future variable, either because there is no need to synchronize with its result, or because the future has been reused and thus the association lost (this does not happen in our example);
- O is the *buffer name* that can be $\mathtt{t}$ or $\mathtt{o}$, which resp. indicate that the task is executing on the same buffer or *maybe* on a different one;
- T can be $\check{m}$, $\tilde{m}$, or $\hat{m}$ where m is a method name. It indicates that the corresponding task is an instance of method m, and its status can be *pending*, *active*, or *finished* resp.;
- P is a natural number indicating the priority of the corresponding task.

Intuitively, an MHP atom $\langle F, O, T, R\rangle$ is read as follows: task T might be executing (in some status) on buffer O with priority P, and one can wait for it to finish using future variable F. The set of all MHP atoms is denoted by $\mathcal{A}$.

Example 4. The MHP atom $\langle x, \mathtt{t}, \tilde{m}, 5\rangle$ indicates that there is an instance of method m running in parallel, in the same buffer. This task is active (i.e., can be at any program point), has priority 5, and is associated with the future x. The MHP atom $\langle\star, \mathtt{o}, \hat{task}, 0\rangle$ indicates that there is an instance of method $task$ running in parallel, maybe in a different buffer. This task is finished (i.e., has executed **return**), has priority 0, and it is associated to any future variable.

An abstract state is a multiset of MHP atoms from $\mathcal{A}$. The set of all multisets over $\mathcal{A}$ is denoted by $\mathcal{B}$. Given $M \in \mathcal{B}$, we write $(a, i) \in M$ to indicate that a appears exactly $i > 0$ times in M. We omit i when it is 1. The local analysis is applied on each method and, as a result, it assigns an abstract state from $\mathcal{B}$ to each program point in the program. The analysis takes into account the priority of the method being analyzed. Thus, since a method might be called with different priorities $\mathtt{p}_1, \ldots, \mathtt{p}_n$, the analysis should be repeated for each $\mathtt{p}_i$. For

the sake of simplifying the presentation, we assume that each method is always called with the same priority. Handling several priorities is a context-sensitive analysis problem that can be done by, e.g., cloning the corresponding code.

The analysis of a given method, with respect to priority $\mathtt{p}$, abstractly executes its code over abstract elements from $\mathcal{B}$. This execution uses a transfer function $\tau_{\mathtt{p}}$, depicted in Fig. 3, to rewrite abstract states. Given an instruction b and an abstract state $M \in \mathcal{B}$, $\tau_{\mathtt{p}}(b, M)$ computes a new abstract state that results from abstractly executing b in state M. Note that the subscript $\mathtt{p}$ in $\tau_{\mathtt{p}}$ is the priority of the method being analyzed. Let us explain the different cases of $\tau_{\mathtt{p}}$:

- (1) Posting a task on the same buffer adds a new MHP atom $\langle y, \mathtt{t}, \check{m}, p\rangle$ to the abstract state. It indicates that an instance of m is pending, with priority p, on the *same* buffer as the analyzed method, and is associated with future variable y. In addition, since y is assigned a new value, those atoms in M that were associated with y should now be associated with $\star$ in the new state. This is done by $M[\langle y, O, Z, R\rangle/\langle\star, O, Z, R\rangle]$ which replaces each atom that matches $\langle y, O, Z, R\rangle$ in M by $\langle\star, O, Z, R\rangle$;
- (2) It is similar to (1), the difference is that the new task might be posted on a buffer different from that of the method being analyzed. Thus, its status should be *active* since, unlike (1), it might start to execute immediately;
- (3)-(5) These cases highlight the use of priorities, and thus mark the main differences wrt [3]. They state that when releasing the processor, only tasks of equal or higher priorities are allowed to become active (simulated through $\mathsf{release}_1$). Moreover, when taking the control back, any task with strictly higher priority is guaranteed to have been finished (simulated through $\mathsf{release}_2$). Importantly, the abstract element after $\mathsf{release}_1$ is associated to the program point of the **release** instruction, and that after $\mathsf{release}_2$ is associated to the program point after the **release** instruction. These two auxiliary instructions are introduced to simulate the implicit "loop" (in the semantics) when the task is waiting at that point;
- (6) This instruction is similar to **release**, the only difference is that the status of the tasks that are associated with future variable y become finished in the following program point. Importantly, the abstract element after $\mathsf{release}_1$ is associated to the program point of the **await** $y?$;
- (7) It changes the status of every pending task executing on the same buffer to active, this is because the processor is released. Note that we do not consider priorities in this case, since the task is finished.

In addition to using the transfer function for abstractly executing basic instructions, the analysis merges the results of paths (in conditions, loops, etc) using a join operator. We refer to [3] for formal definitions of the basic abstract interpretations operators. In what follows, we assume that the result of the local phase is given by means of a mapping $\mathcal{L}_P{:}P_{\mathcal{P}} \mapsto \mathcal{B}$ which maps each program point p (including entry and exit points) to an abstract state $\mathcal{L}_P(p) \in \mathcal{B}$.

Example 5. Applying the local analysis on main, results in the following abstract states (initially the abstract state is $\emptyset$):

$28{:}\{\langle \star, \mathtt{t}, \check{\mathsf{f}}, 10\rangle\}$
$29{:}\{\langle \star, \mathtt{t}, \tilde{\mathsf{f}}, 10\rangle, \langle \mathsf{x}, \mathtt{t}, \tilde{\mathsf{m}}, 5\rangle\}$
$30{:}\{\langle \star, \mathtt{t}, \hat{\mathsf{f}}, 10\rangle, \langle \mathsf{x}, \mathtt{t}, \hat{\mathsf{m}}, 5\rangle\}$
$31{:}\{\langle \star, \mathtt{t}, \hat{\mathsf{f}}, 10\rangle, \langle \mathsf{x}, \mathtt{t}, \hat{\mathsf{m}}, 5\rangle, \langle \star, \mathtt{t}, \check{\mathsf{h}}, 10\rangle\}$
$32{:}\{\langle \star, \mathtt{t}, \hat{\mathsf{f}}, 10\rangle, \langle \mathsf{x}, \mathtt{t}, \hat{\mathsf{m}}, 5\rangle, \langle \star, \mathtt{t}, \check{\mathsf{h}}, 10\rangle\}$
$33{:}\{\langle \star, \mathtt{t}, \hat{\mathsf{f}}, 10\rangle, \langle \mathsf{x}, \mathtt{t}, \hat{\mathsf{m}}, 5\rangle, \langle \star, \mathtt{t}, \check{\mathsf{h}}, 10\rangle, \langle \star, \mathsf{o}, \tilde{\mathsf{task}}, 0\rangle\}$

Note that in the abstract state at program point 30 we have both f and m finished, this is because they have higher priority than main, and thus, while main is waiting at program point 29 both f and m must have completed their execution before main can proceed to the next instruction. If we ignore priorities, then we would infer that f might be active at program point 30 (which is less precise).

5 MHP Graph for Priority-based Scheduling

In this section we will construct a MHP graph relating program points and methods in the program, that will be used to extract precise information on which program points might globally run in parallel. In order to build this graph, we use the local information computed in Sec. 4 which already takes priorities into account. In Sec. 5.2, we explain how to use the MHP graph to infer the MHP pairs in the program. Finally, in Sec. 5.3 we compare the inference method of MHP pairs using a priority-based scheduling with the technique introduced in [3] for programs with a non-deterministic scheduling.

5.1 Construction of the MHP Graph with Priorities

The MHP graph has different types of nodes and different types of edges. There are nodes that represent the status of methods (active, pending or finished) and nodes that represent the program points. Outgoing edges from method nodes are unweighted and unlabeled, they represent points of which at most one might be executing. Outgoing edges from program point nodes are *labeled*, written $\rightarrow_l$ where the label l is a tuple (O, R) that contains a priority R and a buffer name O. These edges represent tasks such that any of them might be running. Besides, when two nodes are directly connected by $i > 1$ edges, we connect them with a single edge superscripted with *weight* i, written as $\rightarrow_l^i$ where l is the label as before.

Definition 2 (MHP graph with priorities). *Given a program P, and its method-level MHP analysis result $\mathcal{L}_P$, the MHP graph of P is a directed graph $\mathcal{G}_P = \langle V, E\rangle$ with a set of nodes V and a set of edges $E = E_1 \cup E_2$ defined:*

$$
\begin{aligned}
V &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \\
E_1 &= \{\tilde{m} \rightarrow p \mid m \in P_{\mathcal{M}}, p \in P_{\mathcal{P}}, p \in m\} \cup \{\hat{m} \rightarrow p_{\check{m}}, \check{m} \rightarrow p_{\hat{m}} \mid m \in P_{\mathcal{M}}\} \\
E_2 &= \{p \rightarrow^i_{(O,R)} x \mid p \in P_{\mathcal{P}}, (\langle \neg, O, x, R\rangle, i) \in \mathcal{L}_P(p)\}
\end{aligned}
$$

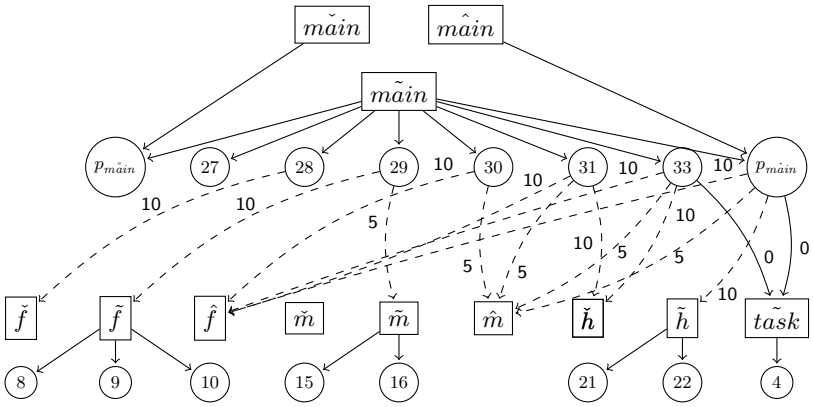

Fig. 4. MHP graph with priorities of the example

Example 6. Fig. 4 depicts the relevant fragment of the MHP graph for our running example. The graph only shows selected program points, namely all points of the `main` task and those points of the other tasks in which there is a **release** instruction, or in which the global memory is updated. For each task, we have three nodes which correspond to their possible status (except for `h` and `task` that we have omitted status that do not have incoming edges). In order to avoid cluttering the graph, in edges from program points, the labels only show the priority. The weight is omitted as it is always 1. The label corresponding to the buffer name is depicted using different types of arrows: normal arrows correspond to the buffer name `o`, while dashed arrows to `t`. From the pending (resp. finished) nodes, we always have an edge to the task entry (resp. exit) point. From the active nodes, we have edges to all program points in the corresponding method body, meaning that only one of them can be executing. The key aspect of the MHP graph is how we integrate the information gathered by the local analysis (with priorities) to build the edges from the program points: we can observe that node 28 has an edge to pending `f`, and at the **await** (node 29) the edges go to active `f` and `m`. After **await**, in nodes 30 and the next ones, the edges go to finished tasks. The remaining tasks only have edges to their program points since they do not make calls to other tasks.

5.2 Inference of Priority-based MHP pairs

The inference of MHP pairs is based on the notion of *intra-buffer path* in the MHP graph. A path from p_1 to p_2 is called *intra-buffer* if the program points p_1 and p_2 are reachable only through tasks in the same buffer. A simple way to ensure the intra-buffer condition is by checking that the buffer labels are always of type `t` (more accurate alternatives are discussed later). Intuitively, two program points $p_1, p_2 \in P_{\mathcal{P}}$ may run in parallel if one of the following conditions hold:

1. there is a non-empty path in $\mathcal{G}_{\mathcal{P}}$ from p_1 to p_2 or vice-versa; or
2. there is a program point $p_3 \in P_{\mathcal{P}}$, and non-empty *intra-buffer* paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$, and the *minimum priority* in both paths is the same; or
3. there is a program point $p_3 \in P_{\mathcal{P}}$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$, and at least one of the paths is not intra-buffer.

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task running from which it is possible to *transitively* reach p_2, or vice-versa. For instance (33,4) is a direct MHP resulting from the direct call from main to task.

The second and third cases correspond to *indirect MHP* scenarios in which a task is running at p_3 and there are two other tasks p_1 and p_2 executing in parallel and both are reachable from p_3. However, the second condition takes advantage of the priority information in intra-buffer paths to discard potential MHP pairs: if the minimum priority of path $pt_1 \equiv p_3 \rightsquigarrow p_1$ is lower than the minimum priority of $pt_2 \equiv p_3 \rightsquigarrow p_2$, then we are sure that the task containing the program point p_2 will be finished before the task containing p_1 starts. For instance, consider the two paths from 29 to 8 and from 29 to 16, which form the potential MHP pair (8,16). They are both intra-buffer (executing on buffer 0) and the minimum priority is not the same (the one to 16 has lower priority). Thus, (16,8) is not an MHP pair. The intuition is that the task with minimum priority (m in this case) will be *pending* and will not start its execution until all the tasks in the other path are finished. Similarly, we obtain that the potential MHP pair (10,15) is not a real MHP pair. Knowing that (10,15) and (16,8) are not MHP pairs is important because this allows us to prove termination of both tasks executing m and f. This is an improvement over the standard MHP analysis in [3], where they are considered as MHP pairs—see Sect. 5.3. On the other hand, when a path involves tasks running in several buffers (condition 3), priorities cannot be taken into account, as the buffers (and their task schedulers) work independently. Observe that, in the second and third conditions, the first edge can only be shared if it has weight $i > 1$ because it denotes that there might be more than one instance of the same type of task running. For instance, if we add the instruction o.task(<>,0) at L33 we will infer the pair (4,4), reporting a potential data race in the access to g2.

Let us formalize the inference of the priority-based MHP pairs. We write $p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$ to indicate that there is a path from p_1 to p_2 in $\mathcal{G}_{\mathcal{P}}$ such that the sum of the edges weights is greater than or equal to 1, and $p_1 \rightarrow^i x \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$ to mark that the path starts with an edge to x with weight i. We will say that a path $p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$ is *intra-buffer* if all the edges from program points to methods have t labels. Similarly, we will say that p is the *lowest priority of the path* $p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}$, written $lowestP(p_1 \rightsquigarrow p_2) = p$, if p is the smallest priority

of all those that appear in edges from program points to methods in the path. We now define the *priority-based MHP pairs* as follows.

Definition 3. *Given a program P, we let $\tilde{\mathcal{E}}_P = D \cup I_{intra} \cup I_{inter}$ where*

$$
\begin{aligned}
D = \{(p_1,p_2) \mid\ & p_1,p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P)\}\\
I_{intra} = \{(p_1,p_2) \mid\ & p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P,\\
& p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \ \textit{is intra-buffer}, \textit{lowestP}(p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1) = pr_1,\\
& p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \ \textit{is intra-buffer}, \textit{lowestP}(p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2) = pr_2,\\
& (x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)) \wedge pr_1 = pr_2\}\\
I_{inter} = \{(p_1,p_2) \mid\ & p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P,\\
& p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \ \textit{or}\ p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \ \textit{are not intra-buffer},\\
& x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\}
\end{aligned}
$$

An interesting point is that even if priorities can only be taken into account at an intra-buffer level, due to the inter-buffer synchronization operations, they allow discarding unfeasible MHP pairs at an inter-buffer level. For instance, we can see that (4,9), which would report an spurious data race, is not an MHP pair. Note that 4 and 9 execute in different buffers. Still, the priority-based local analysis has allowed us to infer that after 29, task `f` will be finished and thus, it cannot happen in parallel with the execution of `task` in buffer `o`. Thus, it is ensured that there will not be a data-race in the access to `g2` from the two different buffers.

The following theorem states the soundness of the analysis, namely, that $\tilde{\mathcal{E}}_P$ is an over-approximation of $\mathcal{E}_P$—the proof appears in the extended version of this paper [6]. Let $\mathcal{E}_P^{non-det}$ be the MHP pairs obtained by [3].

Theorem 1 (soundness). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P \subseteq \mathcal{E}_P^{non-det}$.

As we have discussed above, a sufficient condition for ensuring the intra-buffer condition of paths is to take priorities into account when all edges are labelled with the `t` buffer. However, if buffers can be uniquely identified at analysis time (as in the language of [9]), we can be more accurate. In particular, instead of using `o` to refer to any buffer, we would use the proper buffer name in the labels of the edges. Then, the intra-buffer condition will be ensured by checking that the buffer name along the considered paths is always the same.

In our language, buffers can be dynamically created, i.e., the number of buffers is not fixed a priori and one could have even an unbounded number of buffers (e.g., using `newBuffer` inside a loop). The standard way to handle this situation in static analysis is by incorporating points-to information [17, 15] which allows us to over-approximate the buffers created. A well-known approximation is by buffer creation site such that all buffers created at the same program point are abstracted by a single abstract name. In this setting, we can take advantage of the priorities (and apply case 2 in Def. 3) only if we are sure that an abstract name is referring to a single concrete buffer. As the task scheduler of each buffer works independently, we cannot use knowledge on the priorities to discard pairs if the abstract buffer might correspond to several concrete buffers. The extension of our framework to handle these cases is subject if future work.

5.3 Comparison with non-Priority MHP Graphs

The new MHP graphs with priority information (Sec. 5.1), and the conditions to infer MHP pairs (Sec. 5.2), are extensions of the corresponding notions in [3]. The original MHP graphs were defined as in Def. 2 with the following differences:

- The edges in E_2 do not contain the label *(O,R)* with the buffer name and the priority, but only the weight.
- The method-level analysis $\mathcal{L}_P(p)$ in [3] does not take priorities into account, so after a **release** instruction, pending tasks are set to active. With the method-level analysis in this paper (Sect. 4), tasks with a higher priority in the same buffer are set to finished after a **release** instruction—case (4) in Fig. 3. This generates less paths in the resulting MHP graph with priorities and therefore less MHP pairs.
- In [3], there is another type of nodes (future variable nodes) used to increase the accuracy when the same future variable is re-used in several calls in branching instructions. For the sake of simplicity we have not included future nodes here as their treatment would be identical as in [3].

Regarding the conditions to infer MHP pairs, only two are considered in [3]:

1. there is a non-empty path in $\mathcal{G}_P$ from p_1 to p_2 or vice-versa; or
2. there is a program point $p_3 \in P_\mathcal{P}$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$.

The first case is the same as the first condition in Sect 5.2. The second case corresponds to indirect MHP scenarios and is a generalization of conditions 2 and 3 in Sect 5.2 without considering priorities and intra-buffer paths. With these conditions, we have that the **release** point 22 cannot happen in parallel with the instructions that modify the value of the loop counter `g1` (namely 8 and 15), because there is no direct or indirect path connecting them starting from a program point. However, we have the indirect MHP pairs (10,15) and (16,8), meaning respectively that at the release point of `f` the counter `g1` can be modified by an interleaved execution of `m` and that at the release point of `m` the counter `g1` can be modified by an interleaved execution of `f`. Such spurious interleavings prevent us from proving termination of the tasks executing `f` and `m` and, as we have seen in Sec. 5.2, they are eliminated with the new MHP graphs with priorities and the new conditions for inferring MHP pairs.

6 Implementation in the MayPar System

We have implemented our analysis in a tool called MayPar [4], which takes as input a program written in the ABS language [12] extended with priority annotations. ABS is based on the concurrency model in Sec. 2 and uses the concept of *concurrent object* to realize the concept of task-buffer, such that object creation corresponds to buffer creation, and a method call `o.m()` posts

a task executing m on the queue of object o. Currently the annotations are provided at the level of methods, instead of at the level of tasks. This is because we lacked the syntax in the ABS language to include annotations in the calls, but the adaptation to calls will be straightforward once we have the parser extended.

We have made our implementation and a series of examples available online at `http://costa.ls.fi.upm.es/costabs/mhp`. After selecting an example, the analysis options allow: the selection of the entry method, enabling the option to consider priorities in the analysis, and several other options related to the format for displaying the analysis results and the verbosity level. After the analysis, MayPar yields in the output the MHP pairs in textual format and also optionally a graphical representation of the MHP graph. Besides, MayPar can be used in an interactive way which allows the user to select a line and the tool highlights all program points that may happen in parallel with it.

The examples on the MayPar site that include priority annotations are within the folder priorities. It is also possible to upload new examples by writing them in the text area. In order to evaluate our proposal, we have included a series of small examples that contain challenging patterns for priority-based MHP analysis (including our running example) and we have also encoded the examples in the second experiment of [9] and adapted them to our language (namely we use **await** on futures instead of assume on heap values). MayPar with priority-scheduling can successfully analyze all of them. Although these examples are rather small programs, this is not due to scalability limits of MayPar. It is rather because of the modeling overhead required to set up actual programs for static analysis.

7 Conclusions and Related Work

May-happen-in-parallel relations are of utmost importance to guarantee the sound behaviour of concurrent and parallel programs. They are a basic component of other analyses that prove termination, resource consumption boundness, data-race and deadlock freeness. As our main contribution, we have leveraged an existing MHP analysis developed for a simplistic scenario in which any task could be selected for execution in order to take task-priorities into account. Interestingly, have succeeded to take priorities into account both at the intra-buffer level and, indirectly, also at an inter-buffer level.

To the best of our knowledge, there is no previous MHP analysis for a priority-based scheduling. Our starting point is the MHP analysis for concurrent objects in [3]. Concurrent objects are almost identical to our multi-buffer asynchronous programs. The main difference is that, instead of buffers, the concurrency units are the objects. The language in [3] is data-race free because it is not allowed to access an object field from a different object. Our main novelty w.r.t. [3] is the integration of the priority-based scheduler in the framework. Although we have considered a cooperative concurrency model in which processor release points are explicit in the program, it is straightforward to handle a preemptive scheduling at the intra-buffer level like in [9], by simply adding a release point after posting a new task. If the posted task has higher priority, the active task will

be suspended and the posted task will become active. Thus, our analysis works directly for this model as well. As regards analyses for Java-like languages [14, 7], we have that a fundamental difference with our approach is that they do not take thread-priorities into account nor consider any synchronization between the threads as we do. To handle preemptive scheduling at the inter-buffer level, one needs to assume processor release points at any instruction in the program, and then the main ideas of our analysis would be applicable. However, we believe that the loss of precision could be significant in this setting.

Acknowledgements

This work was funded partially by EU project FP7-ICT-610582 ENVISAGE: *Engineering Virtualized Services* (`http://www.envisage-project.eu`), by the Spanish projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and by the Madrid Regional Government project S2009TIC-1465. We also want to acknowledge Antonio Flores-Montoya for his help and advice when implementing the analysis in the MayPar system.

References

1. Ericsson AB. *Erlang Efficiency Guide*, 5.8.5 edition, October 2011. From `http://www.erlang.org/doc/efficiency_guide/users_guide.html`.
2. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *FORTE'12*, *LNCS 7273*, pages 35–51. Springer, 2012.
4. E. Albert, A. Flores-Montoya, and S. Genaim. Maypar: a May-Happen-in-Parallel Analyzer for Concurrent Objects. In *SIGSOFT/FSE'12*, pages 1–4. ACM, 2012.
5. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*. To appear.
6. E. Albert, S. Genaim, and E. Martin-Martin. May-Happen-in-Parallel Analysis for Priority-based Scheduling (Extended Version). Technical Report SIC 12/13. Univ. Complutense de Madrid, 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. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. M. Emmi, A. Lal, and S. Qadeer. Asynchronous programs with prioritized task-buffers. In *SIGSOFT FSE*, page 48. ACM, 2012.
10. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, Lecture Notes in Computer Science, pages 273–288. Springer, 2013.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
12. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

13. 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.
14. L. Li and C. Verbrugge. A practical mhp information analysis for concurrent java programs. In *LCPC'04*, LNCS, pages 194–208. Springer, 2004.
15. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *ISSTA*, pages 1–11, 2002.
16. M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of ICSE*, pages 386–396. IEEE, 2009.
17. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

A Proof (Sketch)

We consider an auxiliary MHP analysis $\tilde{\mathcal{E}}_P^{loc}$ that is strictly greater than the $\tilde{\mathcal{E}}_P$. Intuitively, the $\tilde{\mathcal{E}}_P^{loc}$ is a refinement over the MHP analysis without priorities [3] where only the method-level analysis has been improved taking priorities into account. Basically, the method-level analysis of $\tilde{\mathcal{E}}_P^{loc}$ is the new one from Figure 3 (page 9) but the graph and the pairs in the graph are defined as in the original MHP analysis in [3].

The structure of our proof is similar to the proof of the analysis without priorities in `http://eprints.ucm.es/16713/`.

Definition 4 (Auxiliary MHP $\tilde{\mathcal{E}}_P^{loc}$). *Given a program P, and its method-level MHP analysis result $\mathcal{L}_P$, the auxiliary MHP graph of P is a directed graph $\mathcal{G}_P^{loc} = \langle V, E\rangle$ with a set of nodes V^{loc} and a set of edges $E^{loc} = E_1^{loc} \cup E_2^{loc}$ defined as:*

$$
\begin{array}{l}
V^{loc} = \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \\
E_1^{loc} = \{\tilde{m} \to p \mid m \in P_{\mathcal{M}}, p \in P_{\mathcal{P}}, p \in m\} \cup \{\hat{m} \to p_{\check{m}}, \check{m} \to p_{\check{m}} \mid m \in P_{\mathcal{M}}\} \\
E_2^{loc} = \{p \xrightarrow{i} x \mid p \in P_{\mathcal{P}}, (\langle \_, \_, x, \_\rangle, i) \in \mathcal{L}_P(p)\}
\end{array}
$$

Given a program P, we let $\tilde{\mathcal{E}}_P^{loc} = D^{loc} \cup I^{loc}$ where

$$
\begin{array}{rl}
D^{loc} = & \{(p_1, p_2) \mid p_1, p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P^{loc})\} \\
I^{loc} = & \{(p_1, p_2) \mid p_1, p_2, p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P^{loc}, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P^{loc}, \\
& \qquad x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\}
\end{array}
$$

We will also consider a modification of the semantics in Figure 1 where states are extended so that tasks inside each buffer contain the additional information $\mathcal{L}_r$ (see Figure 5). Evaluation using the extended semantics will be denoted as $S \rightsquigarrow^r S'$. This set $\mathcal{L}_r$ contains the tasks that have been called in the current task and their status ($\check{tid}$, $\tilde{tid}$ or $\hat{tid}$) as well as the future variable related to them. $\mathcal{L}_r$ is a concrete version of $\mathcal{L}_P$ for each concrete task in a state.

Given a state of this extended semantics, we will define the *concrete graph* of S ($\mathcal{G}_S^r$) and the set of concrete MHP pairs induced by the concrete graph ($\mathcal{EG}_S^r$) as follows:

Definition 5 (Concrete graph and MHP set). *Given a state $S = \langle g, \mathit{buffer}(bid_1, lk_1, \mathcal{Q}_1) \parallel \ldots \mathit{buffer}(bid_n, lk_n, \mathcal{Q}_n)\rangle$, $\mathcal{Q} = \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_n$, the* concrete graph $\mathcal{G}_S^r = \langle V^r, E^r\rangle$ *is defined as:*

$$
\begin{array}{rl}
V_S^r = & \{\tilde{tid}, \hat{tid}, \check{tid} \mid \langle tid, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q}\} \cup cP_S \\
cP_S = & \{(tid, pp(s)) \mid \langle tid, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q}\} \\
E_S = & ei_S \cup el_S \\
ei_S = & \{\tilde{tid} \to (tid, pp(s)) \mid \langle tid, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q}\} \cup \\
& \{\hat{tid} \to (tid, p_{tid}) \mid \langle tid, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q}\} \cup \\
& \{\check{tid} \to (tid, p_{\hat{tid}}) \mid \langle tid, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q}\} \cup \\
el_S = & \{(tid, pp(s)) \to x \mid \langle tid, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q} \wedge (\star : x \in \mathcal{L}_r \vee y : x \in \mathcal{L}_r)\}
\end{array}
$$

$$\text{(NEWBUFFER)}\ \frac{\text{fresh}(bid')\ ,\ l' = l[x \to bid'],\ t = tsk(tid, m, p, l, \langle x = \mathsf{newBuffer}; s\rangle, \mathcal{L}_r)}{\begin{array}{c} buffer(bid, tid, t \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, tid, tsk(tid, m, p, l', s, \mathcal{L}_r) \cup \mathcal{Q}) \parallel buffer(bid', \bot, \{\}) \parallel B \end{array}}$$

$$\text{(PRIORITY)}\ \frac{highestP(\mathcal{Q}) = tid,\ t = tsk(tid, \_, \_, \_, s, \mathcal{L}_r) \in \mathcal{Q},\ s \neq \epsilon(v)}{buffer(bid, \bot, \mathcal{Q}) \parallel B \rightsquigarrow^r buffer(bid, tid, \mathcal{Q}) \parallel B}$$

$$\text{(ASYNC)}\ \frac{\begin{array}{c} l(x) = bid_1,\ \text{fresh}(tid_1),\ l' = l[y \to tid_1],\ l_1 = buildLocals(\bar{z}, m_1) \\ \mathcal{L}_{r'} = \mathcal{L}_r[y : x/\star : x] \cup (y : \check{tid_1}) \end{array}}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \langle y = x.m_1(\bar{z}, p_1); s, \mathcal{L}_r\rangle \cup \mathcal{Q}) \parallel buffer(bid_1, \_, \mathcal{Q}') \parallel B \rightsquigarrow^r \\ buffer(bid, tid, tsk(tid, m, p, l', s, \mathcal{L}_{r'}) \cup \mathcal{Q}) \parallel \\ buffer(bid_1, \_, tsk(tid_1, m_1, p_1, l_1, body(m_1)) \cup \mathcal{Q}') \parallel B \end{array}}$$

$$\text{(AWAIT1)}\ \frac{l(y) = tid_1,\ tsk(tid_1, \_, \_, \_, s_1) \in \mathtt{Buf}, s_1 = \epsilon(v)\mathcal{L}_{r'} = \mathcal{L}_r[y : \check{tid_1}/y : \hat{tid_1}]}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, tid, tsk(tid, m, p, l, s, \mathcal{L}_{r'}) \cup \mathcal{Q}) \parallel B \end{array}}$$

$$\text{(AWAIT2)}\ \frac{l(y) = tid_1,\ tsk(tid_1, \_, \_, \_, s_1) \in \mathtt{Buf}, s_1 \neq \epsilon(v)}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, \bot, tsk(tid, m, p, l, \langle \mathsf{await}\ y?; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \end{array}}$$

$$\text{(RELEASE)}\ \frac{}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \langle \mathsf{release}; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, tid, tsk(tid, m, p, l, \langle \mathsf{release}_1; \mathsf{release}_2; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \end{array}}$$

$$\text{(RELEASE1)}\ \frac{\mathcal{L}_{r'} = \mathcal{L}_r[y : \check{x}/y : \tilde{x}]\ \textit{where}\ priority(x) \geq p}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \langle \mathsf{release}_1; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, \bot, tsk(tid, m, p, l, s, \mathcal{L}_{r'}) \cup \mathcal{Q}) \parallel B \end{array}}$$

$$\text{(RELEASE2)}\ \frac{\mathcal{L}_{r'} = \mathcal{L}_r[y : \tilde{x}/y : \hat{x}]\ \textit{where}\ priority(x) > p}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \langle \mathsf{release}_2; s\rangle, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, \bot, tsk(tid, m, p, l, s, \mathcal{L}_{r'}) \cup \mathcal{Q}) \parallel B \end{array}}$$

$$\text{(RETURN)}\ \frac{v = l(x),\ \mathcal{L}_{r'} = \mathcal{L}_r[y : \check{x}/y : \tilde{x}]}{\begin{array}{c} buffer(bid, tid, tsk(tid, m, p, l, \mathsf{return}\ x;, \mathcal{L}_r) \cup \mathcal{Q}) \parallel B \rightsquigarrow^r \\ buffer(bid, \bot, tsk(tid, m, p, l, \epsilon(v), \mathcal{L}_{r'}) \cup \mathcal{Q}) \parallel B \end{array}}$$

Fig. 5. Extended Semantics

Using the concrete graph, we define the set of concrete MHP $\mathcal{EG}_S^r$:

$$\begin{aligned}\mathcal{EG}_S^r &= dMHP \cup iMHP\\ dMHP &= \{(a,b) \mid a,b \in cP_S \wedge a \rightsquigarrow b \in \mathcal{G}_S^r\}\\ iMHP &= \{(a,b) \mid a,b \in cP_S \wedge (\exists c \in cP_S : c \rightsquigarrow a \in \mathcal{G}_S^r \wedge c \rightsquigarrow b \in \mathcal{G}_S^r)\}\end{aligned}$$

We will also define the set of MHP pairs at runtime:

Definition 6. *Given a program P, we define the set of MHP pairs at runtime as:*

$$\mathcal{E}_P^r = \bigcup\{\mathcal{E}_S^r \mid S_o \rightsquigarrow^r S\}$$

For each state $S = \langle g, buffer(bid_1, lk_1, \mathcal{Q}_1) \parallel \ldots buffer(bid_n, lk_n, \mathcal{Q}_n)\rangle$, $\mathcal{Q} = \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_n$, the set of MHP pairs $\mathcal{E}_S^r$ at runtime is:

$$\mathcal{E}_S^r = \{((tid_1, pp(s_1)), (tid_2, pp(s_2)) \mid \mid \langle tid_1, \_, \_, \_, s, \_\rangle \in \mathcal{Q}, \langle tid_2, \_, \_, \_, s, \_\rangle \in \mathcal{Q}, tid \neq tid_2\}$$

Finally, we define the abstraction function φ over pairs of (*task identifier*, *program point*) to obtain the set of MHP program points $\mathcal{EG}_S$ induced by the concrete graph $\mathcal{G}_S^r$. The abstraction function φ is extended to sets of pairs in the obvious way.

Definition 7 (Abstraction function φ). $\varphi(tid, p) = p$

Definition 8.

$$\begin{aligned}\mathcal{EG}_S &= \{(\varphi(tid_1, p_1), \varphi(tid_1, p_1)) \mid ((tid_1, p_1), (tid_2, p_2)) \in \mathcal{EG}_S^r\}\\ &= \{(p_1, p_2) \mid ((tid_1, p_1), (tid_2, p_2)) \in \mathcal{EG}_S^r\}\end{aligned}$$

Using the previous notions, we can proceed with the proof of Theorem 1.

Theorem 1 (soundness) $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P \subseteq \mathcal{E}_P^{non-det}$.

Proof. To prove the first part of Theorem 1, $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$, we will prove that $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P^{loc}$ (Lemma 1) and $\tilde{\mathcal{E}}_P^{loc} \subseteq \tilde{\mathcal{E}}_P$ (Lemmas 5 and 6). For the second part, $\tilde{\mathcal{E}}_P \subseteq \mathcal{E}_P^{non-det}$, we apply Lemma 9.

Lemma 1 ($\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P^{loc}$).

Proof.

$$\mathcal{E}_P \overset{(1)}{=} \varphi(\mathcal{E}_P^r) \overset{(2)}{=} \varphi(\bigcup_S \mathcal{E}_S^r) \overset{Lemma\ 2}{\subseteq} \varphi(\bigcup_S \mathcal{EG}_S^r) \overset{(3)}{=} \bigcup_S \varphi(\mathcal{EG}_S^r) \overset{(4)}{=} \bigcup_S \mathcal{EG}_S \overset{Lemma\ 3}{\subseteq} \tilde{\mathcal{E}}_P^{loc}$$

The equality (1) holds because the extended semantics (Figure 5) and the semantics from Figure 1 are equivalent w.r.t. the MHP points, since $pp(\textsf{release}) = pp(\textsf{release}_1) = pp(\textsf{release}_2)$. The step marked with (2) is true by Definition 6. Step (3) holds trivially, since it is the application of the abstraction function φ. Finally, equality (3) is true by Definition 8.

Lemma 2. $\forall S : (S_0 \rightsquigarrow^{r*} S) \Rightarrow (\mathcal{E}_S^r \subseteq \mathcal{EG}_S^r)$

Proof. The proof is similar to the proof of Theorem A.1.4 in `http://eprints.ucm.es/16713/`. We have to prove that, given a state S such that $S_0 \rightsquigarrow^{r*} S = \langle g, \mathit{buffer}(\mathit{bid}_1, \mathit{lk}_1, \mathcal{Q}_1) \parallel \ldots \mathit{buffer}(\mathit{bid}_n, \mathit{lk}_n, \mathcal{Q}_n)\rangle$ with $\mathcal{Q} = \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_n$ and $\langle \mathit{tid}_1, \_, \_, \_, s, \_\rangle \in \mathcal{Q}, \langle \mathit{tid}_2, \_, \_, \_, s, \_\rangle \in \mathcal{Q}, \mathit{tid} \neq \mathit{tid}_2$, the set $\mathcal{EG}^r_S$ contains the pair $((\mathit{tid}_1, pp(s_1)), (\mathit{tid}_2, pp(s_2)))$. For that, we will prove that every program point $(\mathit{tid}, pp(s))$ reachable from S_0 using the extended semantics is also reachable from a node $(0, pp(s_0))$ of the main task in the concrete graph $\mathcal{G}^r_S$, i.e., $(0, pp(s_0)) \rightsquigarrow^* (\mathit{tid}, pp(s)) \in \mathcal{G}^r_S$.

The proof proceeds by induction on the length of the derivation $S_0 \rightsquigarrow^* S$. The semantic rules from Figure 5 have different effects on the states, but they can be split into *atomic* steps that maintain the property: *sequential step*, *release*, *loss of a future variable*, *new task added*, *task ending* and *take lock*. Then we express the semantics rules as combinations of these atomic, property preserving steps:

- Rule (NEWBUFFER) is an instance of a *sequential step*.
- Rule (PRIORITY) is an instance of a *take lock* step.
- Rule (ASYNC) is an instance of a *sequential step* followed by a *loss of future variable* and a *new task added* step.
- Rule (AWAIT1) is an instance of a *sequential step* and a *task ending*.
- Rule (AWAIT2) is an instance of a *sequential step*.
- Rule (RELEASE) is an instance of a *sequential step*.
- Rule (RELEASE1) is an instance of a *sequential step* and a *release*.
- Rule (RELEASE2) is an instance of a *sequential step* and a *task ending*.
- Rule (RETURN) is an instance of a *sequential step* and a *release*.

To prove Lemma 3 we need the following definitions:

Definition 9 (Order on MHP atoms). *The set $\mathcal{A}$ is partially ordered as follows: we first let $\check{m}$ and $\hat{m}$ are smaller than $\tilde{m}$ (since it includes both entry and exit program points), and any future variable is smaller than $\star$ (since it includes all future variables), then, we say that $\langle F_1, O_1, T_1, P_2\rangle \preceq \langle F_2, O_2, T_2, P_2\rangle$ iff $O_1 = O_2$, $P_1 = P_2$, F_1 is smaller than or equal to F_2, and T_1 is smaller than or equal to T_2. Given $M_1, M_2 \in \mathcal{B}$, we say that $a \in M_2$* covers *$a' \in M_1$ if $a' \preceq a$. We say that $M_1 \sqsubseteq M_2$ if all elements of M_1 are covered by* different *elements from M_2.*

Definition 10 (Upper-bounds on $\mathcal{B}$). *The join (or upper-bound) of M_1 and M_2 in $\mathcal{B}$, denoted $M_1 \sqcup M_2$, is an operation that calculates a multiset $M_3 \in \mathcal{B}$ such that $M_1 \sqsubseteq M_3$ and $M_2 \sqsubseteq M_3$. Note that it is not guaranteed that the least upper bound exists [3], and thus $\sqcup$ can be defined in several ways. For loops, in order to guarantee termination, if the multiplicity of a given MHP atom* a *increases in each iteration, then it is set to ∞.*

Definition 11 (Abstraction of $\mathcal{L}_r$ sets). *Consider that $S = \langle g, \mathit{buffer}(\mathit{bid}_1, \mathit{lk}_1, \mathcal{Q}_1) \parallel \ldots \mathit{buffer}(\mathit{bid}_n, \mathit{lk}_n, \mathcal{Q}_n)\rangle$, $\mathcal{Q} = \mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_n$ and $\langle \mathit{tid}_1, m, \_, \_, \_, \_\rangle \in \mathcal{Q}$. We define*

the following functions ψ_S, ψ'_S and ψ''_S to obtain multisets in $\mathcal{B}$ from $\mathcal{L}_r$ sets as:

$$\begin{aligned}
\psi''_S(t\check{i}d) &= \check{m}\\
\psi''_S(t\tilde{i}d) &= \tilde{m}\\
\psi''_S(t\hat{i}d) &= \hat{m}\\
\psi'_S(y:x) &= y:\psi''_S(x)\\
\psi_S(\mathcal{L}_r) &= \{(\psi'_S(a),i) \mid a \in \mathcal{L}_r, \#i : b \in \mathcal{L}_r : \psi'_S(a) = \psi'_S(b)\}
\end{aligned}$$

Lemma 3. $\forall S : (S_0 \rightsquigarrow^{r*} S) \Rightarrow (\mathcal{EG}_S \subseteq \tilde{\mathcal{E}}_P^{loc})$

Proof. We have to prove that every pair $(a,b) \in \mathcal{EG}_S$ is also in $\tilde{\mathcal{E}}_P^{loc}$. The proof is a case distinction over the origin of the pair—$(a,b) \in dMHP$ or $(a,b) \in iMHP$—and using the fact that the $\mathcal{L}_P$ obtained for each instruction is more general than the concrete $\mathcal{L}_r$ defined by the extended semantics (Lemma 4).

Lemma 4. $\forall S = \langle g, buffer(bid_1, lk_1, \mathcal{Q}_1) \parallel \ldots buffer(bid_n, lk_n, \mathcal{Q}_n)\rangle$, *if* $S_0 \rightsquigarrow^{r*} S$ *and* $\langle tid_1, m, p, l, s, \mathcal{L}_r\rangle \in \mathcal{Q}$ *then* $\psi_S(\mathcal{L}_r) \sqsubseteq \mathcal{L}_P(pp(s))$.

Proof. It follows easily by induction on the length of the derivation $S_0 \rightsquigarrow^{r*} S$, since the transfer function $\tau_{\mathtt{p}}$ in Figure 3 adds the same information as the rules of the extended semantics of Figure 5. We notice that, for any program point p in a loop, the value of $\mathcal{L}_P(p)$ will be an upper bound of the values of $\tau_{\mathtt{p}}(p)$ after a number of iterations. Therefore, for any iteration in the computation, $\psi_S(\mathcal{L}_r) \sqsubseteq \mathcal{L}_P(p)$.

Lemma 5 $\mathbf{((\tilde{\mathcal{E}}_P^{loc} \setminus \tilde{\mathcal{E}}_P) \cap \mathcal{E}_P = \emptyset)}$**.**

Proof. We have to prove that all pairs that $\tilde{\mathcal{E}}_P^{loc}$ adds over $\tilde{\mathcal{E}}_P$ are not real MHP pairs in $\mathcal{E}_P$, i.e., that the pairs removed by the new global phase of Section 5.2 cannot happen in any computation from S_0. The graph $\mathcal{G}_P$ of Definition 2 and $\mathcal{G}_P^{loc}$ of Definition 4 contain the same set of vertices, and the edges are the same except from the labels (buffer name O and priority P) added in E_2 over E_2^{loc}. The set of MHP pairs is defined as $\tilde{\mathcal{E}}_P^{loc} = D^{loc} \cup I^{loc}$ and $\tilde{\mathcal{E}}_P = D \cup I^{intra} \cup I^{inter}$, but as the graphs are the same $D^{loc} = D$. Extra MHP pairs added by $\tilde{\mathcal{E}}_P^{loc}$ are in $I^{extra} = I^{loc} \setminus (D \cup I^{intra} \cup I^{inter})$. Looking at the definitions of these sets of indirect pairs, it is easy to see that pairs (a,b) in I^{extra} verify:

$$\begin{aligned}
&a, b, c \in P_{\mathcal{P}}, c \xrightarrow{i} a' \rightsquigarrow a \in \mathcal{G}_P, c \xrightarrow{j} b' \rightsquigarrow b \in \mathcal{G}_P,\\
&c \xrightarrow{i} a' \rightsquigarrow a \ \textit{is intra}-\textit{buffer}, lowestP(c \xrightarrow{i} a \rightsquigarrow a) = pr_1,\\
&c \xrightarrow{j} b' \rightsquigarrow b \ \textit{is intra}-\textit{buffer}, lowestP(c \xrightarrow{j} b' \rightsquigarrow b) = pr_2,\\
&(x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)) \wedge \underline{pr_1 \neq pr_2}\},\\
&(a,b) \notin D \cup I^{intra} \cup I^{inter}
\end{aligned}$$

(Notice that a pair may appear in D, I^{intra} or I^{inter} at the same time). Suppose that $pr_1 < pr_2$, the method with minimum priority is m and a, b are not entry points of methods. We proceed by contradiction:

Any evaluation leading to the MHP pair (a, b) must lead to a state $S_0 \rightsquigarrow^* S' = \langle g, \mathit{buffer}(\mathit{bid}_1, \mathit{lk}_1, \mathcal{Q}_1)\rangle \parallel B$, where $\langle \mathit{tid}_1, m_1, \_, \_, \_\rangle \in \mathcal{Q}_1$, $\langle \mathit{tid}_2, m_2, \_, \_, \_\rangle \in \mathcal{Q}_1$, $m_1 = \mathit{method}(pp(a'))$ and $m_2 = \mathit{method}(pp(b'))$. In state S' methods containing program points a' and b' are in the queue. From S', the evaluation must lead to a state $S' \rightsquigarrow^* S$ that generates the MHP pair (a, b), i.e., $S = \langle g, \mathit{buffer}(\mathit{bid}_1, \mathit{lk}_1, \mathcal{Q}_1)\rangle \parallel B$, where $\langle \mathit{tid}'_1, \_, \_, \_, s_1\rangle \in \mathcal{Q}_1$, $\langle \mathit{tid}'_2, \_, \_, \_, s_2\rangle \in \mathcal{Q}_1$, $pp(s_1) = a$ and $pp(s_2) = b$. In the derivation $S' \rightsquigarrow^* S$, the evaluation must pass through a state S'' where method m starts to execute (as a is not an entry point), i.e., $S'' = \langle g, \mathit{buffer}(\mathit{bid}_1, \underline{\mathit{tid}_m}, \mathcal{Q}_1)\rangle \parallel B$, where $\langle \underline{\mathit{tid}_m}, m, \_, \_, \mathit{body}(m)\rangle \in \mathcal{Q}_1$. However, state S'' cannot be reached from S. When the semantic rule (PRIORITY) is applied to give the lock of $\mathcal{Q}_1$ to task tid_m, task tid_2 (or any other task reachable from m_2) will appear in $\mathcal{Q}_1$. Since pr_1 is less than m_2 or any other task reachable from m_2, tid_m could never get the lock.

Lemma 6. $A \subseteq B \wedge (B \setminus C) \cap A = \emptyset \Rightarrow A \subseteq C$

Proof. By contraposition, we prove $A \nsubseteq C \Rightarrow A \nsubseteq B \vee (B \setminus C) \cap A \neq \emptyset$. Since $A \nsubseteq C$, consider $x \in A$ such that $x \notin C$. If $x \in B$ then $x \in (B \setminus C) \cap A \neq \emptyset$. On the other hand, if $x \notin B$ then $A \nsubseteq B$.

To prove that the new set of MHP pairs is smaller than set obtained by the previous MHP analysis in [3] that does not consider priorities ($\tilde{\mathcal{E}}_P \subseteq \mathcal{E}_P^{non-det}$) we will prove that every path between program points in the graph of the new analysis is also a path in the previous graph. (Notice that paths between program points must have an even number of edges: edges from a program point to a method node are followed by other edge from the method node to another program point). We will use the following notions:

Definition 12 ($\mathcal{E}_P^{non-det}$). *Let $\mathcal{L}_P^{nd}(p)$ the method-level MHP analysis result in [3]. The set of MHP pairs $\mathcal{E}_P^{non-det}$ in [3] (without considering the optimization using future nodes) is defined using the graph $\mathcal{G}_P^{nd}$ as follows:*

$$
\begin{aligned}
\mathcal{G}_P^{nd} &= \langle V^{nd}, E^{nd}\rangle \\
E^{nd} &= E_1^{nd} \cup E_2^{nd} \\
V^{nd} &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \\
E_1^{nd} &= \{\tilde{m} \to p \mid m \in P_{\mathcal{M}}, p \in P_{\mathcal{P}}, p \in m\} \cup \{\hat{m} \to p_{\check{m}}, \check{m} \to p_{\hat{m}} \mid m \in P_{\mathcal{M}}\} \\
E_2^{nd} &= \{p \xrightarrow{i} x \mid p \in P_{\mathcal{P}}, (\star : x, i) \in \mathcal{L}_P^{nd}(p)\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}_P^{non-det} &= D^{nd} \cup I^{nd} \\
D^{nd} &= \{(p_1, p_2) \mid p_1, p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P^{nd})\} \\
I^{nd} &= \{(p_1, p_2) \mid p_1, p_2, p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P^{nd}, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P^{nd}, \\
&\quad x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\}
\end{aligned}
$$

Notice that $V^{nd} = V$, $E_1^{nd} = E_1$ from Definition 2 and $D^{nd} = D$ from Definition 3.

Definition 13 (Order of MHP atoms). *We say that an MHP atom with priority and buffer $\langle F, O, T, P\rangle$ is smaller than a MHP atom $F : T$ (from [3]),*

written $\langle F, O, T, P\rangle \preceq' (F' : T')$*, if* F *is smaller than* F' *and* T *is smaller than* T'*. We extend this order to an order between multisets of MHP atoms (*$M \sqsubseteq' M'$*) as in Definition 9.*

Lemma 7. $\forall p \in P_{\mathcal{P}}.\mathcal{L}_{P}(p) \sqsubseteq' \mathcal{L}^{nd}_{P}(p)$

Proof. Straightforward, since the transfer function $\tau_{\mathtt{p}}$ (Figure 3) only differs from the transfer function from [3] (we will write it $\tau^{nd}_{\mathtt{p}}$) in the case of a **release** instruction. As the program point related to a release instruction is the same as the program point of its $\textbf{release}_1$ and $\textbf{release}_2$ instructions, for this case $\tau_{\mathtt{p}}(\textbf{release}, M) = \tau_{\mathtt{p}}(\textbf{release}_1, M) \cup \tau_{\mathtt{p}}(\textbf{release}_2, M')$, where $M' = M[\langle Y, \mathtt{t}, \check{m}, p\rangle / \langle Y, \mathtt{t}, \tilde{m}, p\rangle]$ for those tasks such that $p \geq \mathtt{p}$. Considering the MHP atoms added in the multiset by any of the **release** instruction we have that:

- If $\langle Y, \mathtt{t}, \tilde{m}, p\rangle$ such that $p \geq \mathtt{p}$, then $(Y : \tilde{m})$ will be added by $\tau^{nd}_{\mathtt{p}}$.
- If $\langle Y, \mathtt{t}, \hat{m}, p\rangle$ such that $p \geq \mathtt{p}$, then $(Y : \tilde{m})$ will be added by $\tau^{nd}_{\mathtt{p}}$, which is bigger.

Since both transfer functions start from $\emptyset$ and the way of computing the upper bound is the same, we conclude that $\mathcal{L}_{P}(p) \sqsubseteq' \mathcal{L}^{nd}_{P}(p)$ for every program point p.

Lemma 8. *If* $a, b, c \in P_{\mathcal{P}}$*,* $m \in P_{\mathcal{M}}$ *and* $a \rightarrow^{i}_{(O,R)} m \rightarrow b \rightsquigarrow^{*} c \in \mathcal{G}_{P}$ *then* $a \xrightarrow{j} m \rightarrow b \rightsquigarrow^{*} c \in \mathcal{G}^{nd}_{P}$*, with* $j \geq i$*.*

Proof. By induction on the length of the path $b \rightsquigarrow^{*} c$.

- Base Case: $a \rightarrow^{i}_{(O,R)} m \rightarrow c \in \mathcal{G}_{P}$
  By Lemma 7 we know that $a \xrightarrow{j} c \in E^{nd}_{2}$ and $j \geq i$. The edge $m \rightarrow c$ is in E_1, so by definition it is also in E^{nd}_{1}.
- Inductive Step: $a \xrightarrow{j} m \rightarrow b \rightsquigarrow^{+} c \in \mathcal{G}_{P}$
  By the Inductive Hypothesis we have that $b \rightsquigarrow^{+} c \in \mathcal{G}^{nd}_{P}$. The reasoning is similar to the previous case: Lemma 7 and $E^{nd}_{1} = E_1$. Therefore, we can construct a path $a \xrightarrow{j} m \rightarrow b \rightsquigarrow^{+} c \in \mathcal{G}^{nd}_{P}$ such that $j \geq i$.

Lemma 9 $(\tilde{\mathcal{E}}_P \subseteq \mathcal{E}^{non-det}_{P})$**.**

Proof. Straightforward by Lemma 8.