A Memory-Optimal Buffering Protocol for Preservation of Synchronous Semantics under Preemptive Scheduling

Christos Sofronis *
Verimag Laboratory
Centre Equation, 2, avenue de Vignate, 38610 Gières, France
sofronis@imag.fr

Stavros Tripakis
CNRS/Verimag and Cadence Design Systems
1995 University Ave, Suite 460, Berkeley, CA 94704, USA.
tripakis@imag.fr

Paul Caspi
Centre Equation, 2, avenue de Vignate, 38610 Gières, France
CNRS/Verimag
caspi@imag.fr

ABSTRACT

Recently, we have proposed a set of buffering schemes to preserve the semantics of a synchronous program when the latter is implemented as a set of multiple tasks running under preemptive scheduling. These schemes, however, are not optimal in terms of memory (buffer usage). In this paper we propose a new protocol which generalizes the previous schemes. The new protocol is not only semantics-preserving but also memory-optimal in two senses: first, in terms of the number of buffers required to preserve semantics in the worst case (i.e., for the "worst" possible arrival/execution pattern of the tasks); second, in terms of the number of buffers required to preserve semantics for any arrival/execution pattern and at any time, assuming no knowledge of future arrivals.

Categories and Subject Descriptors

D2.2.2 [**Software**]: Software EngineeringDesign Tools and Techniques; D2.2.3 [**Software**]: Software EngineeringCoding Tools and Techniques

General Terms

Algorithms, Design, Performance, Reliability

Keywords

embedded software, model-based design, optimality, preemptive scheduling, process communication, semantical preservation, synchronous programming

*This work has been supported by a project of the Region Rhone-Alpes, France.

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.
EMSOFT'06, October 22–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-542-8/06/0010 ...$5.00.

1. INTRODUCTION

The *model-based design* paradigm advocates the use of models with formal or semi-formal semantics through-out the entire development cycle, from design to implementation. One or more models of the system are built using one or more modeling languages of choice (different models may be necessary to capture different parts or "views" of the system). These models are subjected to simulation or even exhaustive verification. This allows to find and correct bugs early enough, which is a crucial cost-savings factor. "What-if" analysis can also be performed at the model level, exploring different design choices.

At some point the implementation phase begins, during which the system is actually built. By "system" we mean hardware, software or both. In the hardware industry the implementation phase is closely coupled with the modeling and analysis phase. Powerful EDA tools, stemming from a rich body of research on logic synthesis and similar topics, are used for gate synthesis, circuit layout, routing, etc. Such tools are largely responsible for the proliferation of electronics and their constant increase in performance.

In the software industry the situation is not as clear. On one hand, high-level models are not as widespread. After all, the software itself *is* a model and simulation can be done by executing the software. Testing and debugging are commonplace (in fact, very time-consuming) but they are done at the implementation level, that is, on the target software. Implementation is automated using the most classical tools in computer science: compilers. The situation is changing, however: languages such as Matlab/Simulink[1], UML[2] and others, as well as corresponding software-synthesis facilities are used more and more. Currently, software synthesis is mostly restricted to separate code generation of parts of the system. The integration of the different pieces of code is usually done "manually" and is source of many problems, since the implementation often exhibits unexpected behavior: deadlocks, missed data values, etc. These problems arise because the implementation method (in this case, code generation followed by manual integration) does not guarantee that the original behavior (high-level semantics) is preserved.

In this context, our work aims at developing implementation methods that guarantee semantical preservation. We

[1]Matlab and Simulink are trademarks of the Mathworks Inc.: http://www.mathworks.com.
[2]of the Object Management Group: http://www.uml.org/

focus on embedded software, and in particular embedded control software, used extensively in domains such as automotive, avionics and others. Wide-spread high-level design languages for embedded control applications are Simulink and synchronous languages [3]. Both Simulink and synchronous languages are based on a *synchronous* or *zero-time* semantics, which means that the system is fast enough to cope with changes in the environment. This semantics allows the designer to abstract from implementation details and focus on the design logic. It also ensures platform-independence which is crucial for portability.

Naturally, the zero-time assumption breaks down upon implementation. This often results in implementations which do not preserve the synchronous semantics. In turn, the results obtained by analyzing the model (e.g., model satisfies a given property) may not hold at the implementation level. Then, in order not to have to repeat the simulation and verification effort at the implementation level, the following issue needs to be addressed: how can the semantics of the high-level model be preserved while relaxing the ideal semantical assumptions?

This question admits different answers depending on what type of implementations are considered. By "implementation" we mean the entire executing system, including computing hardware, network, operating system(s), middleware and application software. "Classic" implementations of synchronous languages are single-processor, single-task implementations [9]. Such implementations consist of a single program of the form:

```
initialize state;
while (true) do {
  await trigger;
  read inputs;
  compute outputs and next state;
  write outputs and update state;
}
```

The `trigger` in the pseudo-code above can be the "tick" of a periodic clock, an interrupt of some external device, etc. In any case, in order to guarantee preservation of the synchronous semantics it suffices to ensure that the worst-case execution time (WCET) of body of the above loop is smaller than the minimum inter-arrival time (MIT) of the `trigger`.

In this paper we are interested in single-processor, *multi-task* implementations, where the tasks are scheduled using a real-time operating system (RTOS) that employs a *preemptive scheduling* policy. Such implementations are useful in the case of *multi-rate* synchronous programs, where different parts of the program are executed at different rates. In such a case, the single-task implementation method above forces the generation of code which is triggered at the fastest rate: this might result in an impossibility to meet the WCET $<$ MIT condition above. For example, consider a synchronous program consisting of two blocks, A and B, such that A must be executed every 10 ms and B every 50 ms. Suppose the WCETs of A and B are 2 ms and 10 ms, respectively. The MIT of `trigger` is 10 ms (the rate of A) while the WCET of the loop is 12 ms, thus, the single-task implementation method cannot be applied in this case.

On the other hand, a multi-task implementation could, in-principle, be used: notice that the processor utilization in the above example is low ($\frac{2}{10} + \frac{10}{50} = 40\%$), thus, schedulability under a preemptive scheduling policy such as *rate-monotonic* is guaranteed [13]. There is, however, an issue of semantical preservation: this is because a multi-task implementation requires inter-task communication (data exchanged between tasks) and it is not obvious how this should be implemented. As shown in [16, 19], "naive" inter-process communication schemes do not preserve the synchronous semantics (see also Section 3.2 for an example). In particular, such schemes are not *deterministic*: depending on the execution time of the tasks, the data sent from a task to another might be different.

One might say that strict preservation of the synchronous semantics is not really necessary. After all, in control applications controllers are usually designed to be *robust* to various types of data variability, including data loss, jitter, sensor inaccuracies, etc. Two answers can be given to this claim. First, controllers contain more and more "discrete logic", which is not robust (a single bit-flip may change the course of an if-then-else statement). Second, echos from the industry indicate that determinism is an important requirement. For instance, recent versions of the Simulink code-generator *Real Time Workshop* provide options to "ensure deterministic data transfer" (see the "Related work" section for references). Our contacts with Esterel Technologies[3] reveal similar concerns.

In [16, 19] we proposed a set of buffering schemes that permit the preservation of the synchronous, zero-time semantics of a set of tasks running under preemptive scheduling. These schemes were developed for a single writer/reader pair of tasks. Although they can also be applied to any number of tasks by considering each writer/reader pair independently, this generally results in a non-optimal usage of buffers. According to Esterel Technologies, this is a big concern. In this paper, we address this problem. We propose a buffering protocol which generalizes those schemes to any number of readers. This protocol is called *dynamic buffering protocol*, or DBP. We prove that DBP is *optimal* in terms of buffer usage, in two senses.

- We provide a set of lower bounds on the number of buffers required in order for the semantics to be preserved. We show that DBP never uses more buffers than what is specified by these lower bounds. This means that DBP is optimal in the worst-case, that is, for the "worst" possible scenario.

- We also show that DBP satisfies a stronger optimality property, namely, that for every task graph, for every arrival/execution pattern of the tasks, and at any time t, the values stored by DBP at time t are precisely those values necessary in order to preserve the semantics, assuming a *non-clairvoyant* buffering protocol, that is, a protocol that is not aware of the future arrival pattern of tasks.

DBP consists of a set of buffers shared by the writer and reader tasks, a set of pointers pointing to these buffers and the protocol to manipulate buffers and pointers. The essential idea behind the protocol is that *the pointers must be manipulated not during the execution of the tasks, but upon the arrival of the events triggering these tasks*. In that way, the order of arrivals can be "memorized" and the original semantics can be preserved. DBP is dynamic in the sense

[3]http://www.esterel-technologies.com/

that it allocates buffers only when they are needed, upon task arrivals. It thus tries to *reuse* those buffers that store values that are no longer needed.

DBP can be applied when the tasks are scheduled using one of the following two preemptive scheduling policies: *static-priority* (SP) or *earliest-deadline first* (EDF). For simplicity, we only consider the SP case in the rest of the paper. The extension to EDF can be done easily using the same idea as in [19]. It should be emphasized that the protocol works *no matter what the arrival pattern of tasks is.* In particular, it works for both *time-triggered* (e.g., multi-periodic) and *event-triggered* applications.

Related work

The Simulink documentation claims that the *Real Time Workshop* code generator is able to provide implementations that reproduce the deterministic behavior of the model, provided tasks are periodic and periods are multiples of each other.[4] The documentation does not describe how this is achieved, however, evidence can be found in some restrictions imposed on *multi-rate* Simulink diagrams. For instance, Simulink requires that a *unit-delay* block be inserted between a "slow" (typically low-priority) writer and a "fast" (typically high-priority) reader. Typically, the "slow" writer will also have a lower priority than the reader, according to the *rate monotonic* priority scheme [13]. In this case, the above restriction avoids the problem where the reader preempts the writer before the latter has time to write. Simulink also requires that a *zero-order-hold* block be inserted between a "fast" writer and a "slow" reader. This avoids the problem where the higher-priority writer preempts the reader before the latter has time to read.

An approach similar to ours is followed in the independent work by Baleani et al. [2]. We extensively comment on this paper since, to our knowledge, it is the closest to our work. There are several differences between [2] and our work. Regarding the setting:

- We work with a known scheduling policy (static-priority or EDF). In [2], no assumption is made about the scheduling policy.
- DBP requires no knowledge of the minimum inter-arrival time (MIT) or deadline of tasks. However, we do require schedulability, namely, that each time a new instance of a task arrives, its previous instance has finished execution. It is beyond the scope of our work to ensure schedulability: the user of our framework must guarantee this, using methods from real-time scheduling theory (e.g., see [13, 10, 1, 18]), more recent methods based on timed automata model-checking (e.g., see [8]), etc. In [2], it is assumed that the MIT and deadline of each task is known. Schedulability is also assumed, in the sense that the deadline of each task is respected. The deadline of a task can be greater than its MIT, which means that the schedulability assumption in [2] is weaker than ours.
- We assume that all tasks run on a single processor. In [2], no a-priori assumption is made about the execution platform.

[4] See the section titled "Models with Multiple Sample Rates" of the Real-Time Workshop user guide, available at www.mathworks.com/access/helpdesk/help/toolbox/rtw/ug/.

The main focus of [2] is to provide upper bounds on the memory required so that the semantics are preserved. Less emphasis is given on the development of communication schemes that achieve these memory bounds. Regarding the memory requirements:

- DBP uses buffers optimally.
- The bounds provided in [2] are generally not tight. For example, in a SP setting with one writer and $N = 2$ readers, where both readers have lower priority than the writer, where the periods are $T_w = 2, T_1 = 3, T_2 = 5$, and the deadlines are equal to the periods, we require 3 buffers whereas the upper bound calculated by the formulas provided in [2] is 4.

Finally, regarding the communication schemes:

- DBP is *lock-free*, meaning that tasks do not block on reads or writes: the only thing that can suspend execution of a task is preemption by another, higher-priority task. DBP requires *atomic* manipulations of global pointers upon task releases. These can be handled by the operating system or by some special interrupt-handling routine with the highest priority.
- [2] considers single-processor, "multi-processor" (many processors with centralized pointer manipulations) or "distributed" (many processors with decentralized pointer manipulations) implementations. For single-processor implementations, both lock-free and locking methods are considered. In the lock-free methods, the assumption is that scheduling ensures that the producer has always higher priority than the consumer. This implies that there can be no cycles in the graph of producers/consumers and that this graph is topologically ordered in order to assign the priorities. The main impact of this assumption is that a consumer task which comes after a long chain of producers will have a very low priority, thus, it cannot handle "urgent" events. In contrast, in our work we make no assumption on the relative priorities of producer and consumer, provided that a unit delay is present when the producer has lower priority than the consumer.

In summary, one can say that the setting of [2] is more general than ours, however, it also requires more knowledge (minimum inter-arrival times, deadlines). Our protocol has optimal memory requirements while the upper bounds provided in [2] are not tight (this is to be expected given that their setting is more general). Our implementation is for a single processor with SP scheduling and any assignment of priorities. The single-processor implementation of [2] assumes that producers have higher priority than consumers. [2] considers also multi-processor implementations.

Related to our work is [12], which considers the *synchronous dataflow* model (SDF) and provides methods for static scheduling and code generation on single-processor or multi-processor architectures. This work has been extended in a number of directions, including buffer optimizations (e.g., see [5, 14]). SDF can be viewed as a subclass of the model we consider in this paper, in the sense that only multi-periodic designs can be described in SDF. On the other hand, SDF descriptions are more high-level and must generally be *unfolded* into a more basic model such as ours.

A major difference with our work, however, is that [12, 5, 14] aim for static, cyclic schedules, whereas we aim for multi-task applications that use dynamic, preemptive scheduling.

Another related paper is [15]. This paper considers the problem of minimizing the cost of adding unit-delay or zero-order-hold blocks in Simulink diagrams. The main difference with our work is that the focus of [15] is cost minimization and not preservation of semantics. Indeed, by adding or removing blocks as the above, the semantics generally change. In contrast, we start from a fixed set of such blocks and do not attempt a modification. We only perform optimizations at the implementation level and not at the design level.

Our work is also related to a set of papers that propose lock-free inter-task communication schemes, for instance [7, 11]. Although DBP is a lock-free protocol (only manipulations of pointers are atomic, writes and reads need not be), it is different from the protocols proposed in the above works. The latter preserve the integrity and often also the "freshness" of data, meaning that the reader consumes the latest complete value produced by the writer. This value does not always correspond to the value defined by the zero-time semantics. Another difference is that DBP is based on pointer manipulations that happen upon task release, and not task execution, as is the case of the above protocols.

We should also note that our work has different objectives from the research done in the context of real-time scheduling theory (e.g., see [13, 10, 1, 18]). Real-time scheduling theory is concerned with checking schedulability of a set of tasks in various settings. Our concern is not schedulability, but preservation of semantics. We *assume* that the system is schedulable (something that can be checked using existing scheduling techniques such as those in the works cited above) and we develop preservation schemes that rely on this assumption.

Organization of this paper

The rest of the paper is organized as follows. Section 2 presents the task model with ideal semantics. Section 3 discusses the execution under static-priority scheduling and issues of semantical preservation during implementation. Section 4 presents DBP (a correctness proof of the protocol is provided the Appendix). In Section 5 we prove that DBP is optimal in terms of buffer utilization. Section 6 presents the conclusions and future work directions.

2. A SYNCHRONOUS INTER-TASK COMMUNICATION MODEL

We consider a set of *tasks*, $\mathcal{T} = \{\tau_1, \tau_2, ...\}$. The set need not be finite, which allows the modelling of, for example, dynamic creation of tasks. In this paper, we do not consider the problem of decomposing a Simulink design or synchronous program to a set of tasks. One method to do this for multi-periodic applications is to group all blocks with the same period in a single task. Notice, however, that our protocol is not restricted to multi-periodic applications: it can work also under event-triggered applications.

To model inter-task communication, we consider a set of *data-flow links* of the form (i, j, p), with $i, j \in \{1, 2, ...\}$ and $p \in \{-1, 0\}$. If $p = 0$ then we write $\tau_i \to \tau_j$, otherwise, we write $\tau_i \stackrel{-1}{\to} \tau_j$. The tasks and links result in what we shall call a *task graph*. For each i, j pair, there can only be one link, so we cannot have both $\tau_i \to \tau_j$ and $\tau_i \stackrel{-1}{\to} \tau_j$.

Intuitively, a link (i, j, p) means that task τ_j receives data from task τ_i. If $p = 0$ then τ_j receives the last value produced by τ_i, otherwise, it receives the one-before-last value (i.e., there is a "unit delay" in the link from τ_i to τ_j). In both cases, it is possible that the first time that τ_j occurs[5] there is no value available from τ_i (either because τ_i has not occurred yet, or because it has occurred only once and $p = -1$). To cover such cases, we will assume that for each task τ_i there is a *default output* value y_0^i. Then, in cases such as the above, τ_j uses this default value.

Notice that links model data-flow, and not *precedences* between tasks.

We allow for cycles in the graph of links, provided these cycles are not *zero-delay*, that is, provided there is at least one link $(i, j, -1)$ in every cycle. Notice that we could allow zero-delay cycles if we made an assumption on the arrival patterns of tasks, namely, that all tasks in a zero-delay cycle cannot occur at the same time. However, it is often the case that many tasks occur at the same time, for instance, in the multi-periodic case where tasks have the same initial phase.

Synchronous, zero-time semantics

We associate with this model an "ideal", *zero-time* semantics. For each task τ_i we associate an *increasing sequence of occurrence times* $T_i = t_1^i < t_2^i < \cdots$, where $t_k^i \in R_{\geq 0}$. We denote t_k^i by $T_i(k)$. Because of the zero-time assumption, the occurrence time captures the release, start and finish times of a task. In the next section, we will distinguish these three times.

We make no assumption on the occurrence times of a task. This allows us to capture all possible situations, namely, where a task is *periodic* (i.e., released at multiples of a given period) or *event-triggered* (i.e., released upon occurrence of an external, generally unpredictable, event). Also note that for two tasks i and j, we might have $t_k^i = t_m^j$, which means that i and j may occur at the same time. The absence of zero-delay cycles ensures that the semantics will still be well-defined in such a case.

Given time $t \geq 0$, we define $n_i(t)$ to be the number of times that τ_i has occurred before t, that is:

$$n_i(t) = \sup\{k | T_i(k) \leq t\}$$

where the sup of the empty set is taken to be 0, so that if τ_i has not occurred before t then $n_i(t) = 0$.

We denote inputs of tasks by x's and outputs by y's. Let y_k^i denote the output of the k-th occurrence of τ_i. Given a link $\tau_i \to \tau_j$, $x_k^{i,j}$ denotes the input that the k-th occurrence of τ_j receives from τ_i. The ideal semantics specifies that this input is equal to the output of the last occurrence of τ_i before τ_j, that is:

$$x_k^{i,j} = y_\ell^i, \text{ where } \ell = n_i(t_k^j).$$

Notice that if τ_i has not occurred yet then $\ell = 0$ and the default value y_0^i is used.

If the link has a unit delay, that is, $\tau_i \stackrel{-1}{\to} \tau_j$, then:

$$x_k^{i,j} = y_\ell^i, \text{ where } \ell = \max\{0, n_i(t_k^j) - 1\}.$$

An example of the ideal semantics is provided in Figure 1, discussed in the next section.

[5]As we shall see shortly, we define an "ideal" zero-time semantics where a task executes and produces its result as the same time it is released. We can thus say "task τ_i occurs".

3. PREEMPTIVE MULTI-TASK IMPLEMENTATIONS

We consider the situation where tasks are implemented as stand-alone processes executing on a single processor equipped with a real-time operating system (RTOS). The RTOS implements a scheduling policy to determine which of the ready tasks (i.e., tasks released but not yet completed) is to be executed at a given point in time. In this paper, we consider a *static-priority* scheduling policy. According to that, each task τ_i is assigned a unique *priority* p_i. The task with the *highest* (greatest) priority among the ready tasks executes. We assume no two tasks have the same priority, that is, $i \neq j \Rightarrow p_i \neq p_j$. This implies that at any given time, there is a unique task that may be executed. In other words, the scheduling policy is *deterministic*, in the sense that for a given pattern of release and execution times, there is a unique behavior.

We should point out that our protocol can also be applied to the case where the RTOS implements *earliest-deadline first* (EDF) scheduling. This can be done using the same idea as the one employed in [19]: instead of considering the relative priorities of tasks, consider their relative *deadlines*. For simplicity of the presentation, we restrict ourselves to the static-priority case and leave the details on the extension to EDF for the full version of this paper.

In the ideal semantics, task execution takes zero time. In reality, this is not true. A task is released and becomes ready. At some later point it is chosen by the scheduler to execute. Until it completes execution, it may be preempted a number of times by other tasks. To capture this, we distinguish the release time of a task τ_i from the time τ_i begins execution and from the time τ_i ends execution. For the k-th occurrence of τ_i, these three times will be denoted r_k^i, b_k^i and e_k^i, respectively.

3.1 A "simple" implementation

Our purpose is to implement the set of tasks so that the ideal semantics are preserved by the implementation. Obviously, different implementation choices exist: simple ones involving a single buffer between each pair of writer/reader tasks, or more sophisticated ones involving, for instance, a *priority inheritance* protocol to avoid the phenomenon of *priority inversion* [17], or a lock-free inter-task communication scheme like the ones mentioned earlier [7, 11]. None of these implementations, however, guarantees preservation of the original synchronous semantics. This has been illustrated through a number of examples provided in [16, 19]. For the sake of understanding, we borrow one of these examples and repeat it here.

The example is based on a "simple" implementation. What we call simple implementation is a buffering scheme where, for each link $\tau_i \rightarrow \tau_j$, there is a buffer $B_{i,j}$ used to store the data produced by τ_i and consumed by τ_j. This buffer must ensure data integrity: a task writing on the buffer might be preempted before it finishes writing, leaving the buffer in an inconsistent state. To avoid this, we will assume that the simple implementation scheme uses *atomic* reads and writes, so that a task writing to or reading from a buffer cannot be preempted before finishing.

For the purposes of this section, we assume that each task is implemented in a way such that all reads happen right after the beginning and all writes happen right before the end of the execution of the task. Also, there is only one read/write per pair of tasks, that is, if $\tau_i \rightarrow \tau_j$ then τ_i cannot write twice to τ_j. These assumptions are not part of the implementation scheme. They are a "programming style". Our aim is to show that, even when this programming style is enforced, the ideal semantics are not generally preserved. Note that these assumptions are not required in the case of DBP: the latter works even when these assumptions do not hold. However, we will assume that every writer task writes at least once at each occurrence. This is not a restrictive assumption since "skipping" a write amounts to memorizing the previously written value (or the default output) and writing this value.

Also, we will assume that the set of tasks is *schedulable*. This means that no task ever violates its absolute deadline, where absolute deadline is the next release time of the task. For example, the absolute deadline of the k-th occurrence of τ_i is r_{k+1}^i.

Obviously, schedulability depends on the assumptions made on the release times and execution times of tasks. Checking schedulability is beyond the scope of this paper. A large amount of work exists on schedulability analysis techniques for different sets of assumptions: see, for instance, the seminar paper of Liu and Layland [13] or the books [10, 18]. Notice, however, that our assumption of schedulability is not related to a specific schedulability analysis method: it cannot be, since we make no assumptions on release times and execution times of tasks.

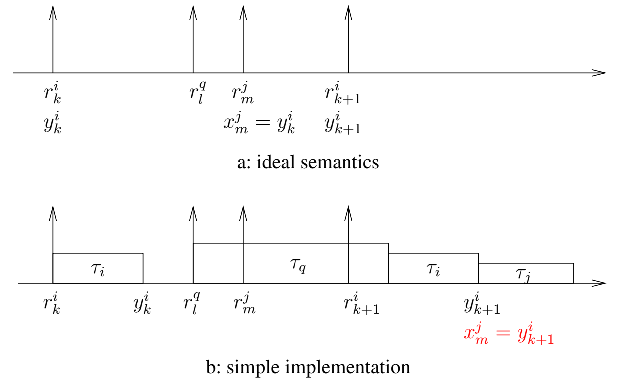

Figure 1: In the semantics, $x_m^j = y_k^i$, whereas in the implementation, $x_m^j = y_{k+1}^i$.

3.2 Problems with the "simple" implementation

Let us now borrow an example from [19] that shows that the synchronous semantics are not always preserved. Consider the case where the writer has higher priority than the reader. In particular, we have $\tau_i \rightarrow \tau_j$ and $p_i > p_j$. There is also a third task τ_q with higher priority than both τ_i and τ_j, $p_q > p_i > p_j$. Consider the scenario shown in Figure 1. The top of the figure shows an arrival pattern of the three tasks. At the bottom a possible execution pattern is shown. We can see that, according to the semantics, the input of the m-th occurrence of τ_j is equal to the output of the k-th occurrence of τ_i. However, this is not true in the implementation. This is because τ_q "masks" the order of arrival of τ_j and τ_i ($r_m^j < r_{k+1}^i$). As a result, the order of execution of

τ_j and τ_i is reversed and the reader τ_j consumes a "future" (according to the semantics) output of the writer τ_i.

4. THE DYNAMIC BUFFERING PROTOCOL

In this section we present the dynamic buffering protocol, or DBP. DBP generalizes the one-writer/one-reader protocols proposed in [16, 19] to any number of readers (it can also be used for any number of writers as explained in Section 4.2). In particular, DBP is used for one writer communicating (the same) data to N lower-priority readers and M higher-priority readers, as shown in Figure 2. In N_1 among the N lower-priority readers there is no unit-delay, while in the rest $N_2 = N - N_1$ readers there is a unit-delay. Notice that for all the higher-priority readers there is a unit-delay: this is because, as shown in [16, 19], it is otherwise impossible to guarantee preservation of the synchronous semantics. Unit-delays are denoted with "-1" in the figure.

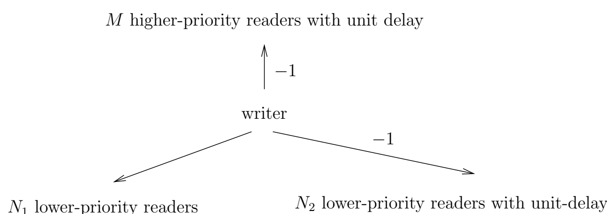

Figure 2: Setting of the DBP protocol.

4.1 The dynamic buffering protocol

The protocol DBP is shown in Figure 3. The figure shows the protocol in the case where $M \neq 0$ or $N_2 \neq 0$, that is, the case where there are links with unit-delay. If $M = N_2 = 0$ then the protocol is actually simpler: the pointer `previous` is not needed and instead of $N + 2 = N_1 + 2$, only $N_1 + 1$ buffers are needed.

The operation of DBP is as follows. The writer τ_w maintains all buffers and pointers except the pointers of the higher-priority readers `P[i]`. The `current` pointer points to the position that the writer last wrote to. The `previous` pointer points to the position that the writer wrote to before that. `R[i]` points to the position that τ_i must read from.

The key point is that when the writer is released, a *free* position in the buffer array must be found, and this is where the writer must write to. By free we mean a position which is not currently in use by any reader, as defined by the predicate `free(j)`. Finding a free $j \in [1..N + 2]$ amounts to finding some j which is different from `previous` (because `B[previous]` may be used by the higher-priority reader or a possible lower-priority with unit-delay reader may need to copy its value) and also different from all `R[i]` (because `B[R[i]]` is used by the lower-priority reader τ_i). Notice that such a j always exists, by the *pigeon-hole principle*: there are $N + 2$ possible values for j and up to $N + 1$ possible values for `previous` and all `R[i]`.

Finding a free position is done in the second instruction executed upon the release of the writer. The first instruction updates the `previous` pointer. This pointer is copied by each higher-priority reader τ_i' into its local variable `P[i]`. τ_i' then reads from `B[P[i]]`.

When a lower-priority reader τ_i is released, we have two cases: (i) either τ_i is one of the N_1 readers that are linked without a unit-delay, or (ii) τ_i is one of the N_2 readers that are linked with unit-delay. In case (i) τ_i needs the last value written by the writer. In case (ii) τ_i needs the previous value. Pointer `R[i]` is set to the needed value. Besides this pointer assignment the rest of the procedure remains the same for both kinds of lower-priority readers. While executing, τ_i reads from `B[R[i]]`. When τ_i finishes execution, `R[i]` is set to `null`. This is done for optimization purposes, so that buffers can be re-used as early as possible. Notice that even if this operation is removed, DBP will still be correct and it will use at most $N + 2$ buffers. However, DBP will be sub-optimal, in the sense that the buffer pointed to by `R[i]` will not be freed until the next release of τ_i. With the above operation present, the buffer is freed earlier, namely, when the current release of τ_i finishes.

Notice that DBP relies on the fact that no more than one instance of every task is active at any point in time. This follows from the schedulability assumption which states that when a task is released all previous instances of this task have finished. Having at most one instance of every reader means that each reader needs at most one buffer to read data from.

We should also note that in case of simultaneous release of writer and one or more readers (e.g., in a multi-periodic application) the release action for the writer (that is, the set of statements corresponding to the writer's release event) is executed first. The release actions for the readers are executed afterwards, in any order.

A proof of correctness of DBP is provided in Appendix A.

An example

To illustrate how DBP works, we provide an example. Consider the task graph shown in Figure 4. There are four tasks: one writer τ_w with period $T_w = 2$, one higher-priority reader τ_1 with period $T_1 = 1$ and two lower-priority readers τ_2 and τ_3 with periods $T_2 = 3$ and $T_3 = 5$ respectively. This means that for the one writer of this task graph $N = 2$, where N is the number of lower priority readers. Moreover there is one task with higher priority. Suppose the priorities of the tasks follow the rate-monotonic assignment policy (note that DBP does not require this, as it can work with any priority assignment):

$$Prio_1 > Prio_w > Prio_2 > Prio_3.$$

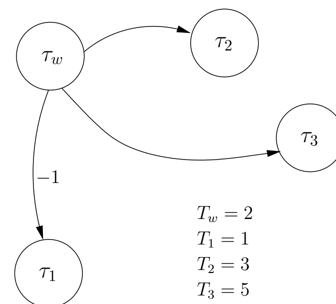

Figure 4: A task graph with one writer and three readers.

Inter-task communication:
$\tau_w \to \tau_i$, for $i = 1, ..., N_1$, $\tau_w \overset{-1}{\to} \tau_i$, for $i = N_1+1, ..., N_1+N_2$, and $\tau_w \overset{-1}{\to} \tau_i'$, for $i = 1, ..., M$. Let $N = N_1+N_2$.

Task τ_w maintains a buffer array `B[1..`N`+2]`, one pointer array `R[1..`N`]` and two pointers `current` and `previous`.
Each task τ_i', for $i = 1, ..., M$, maintains a local pointer `P[i]`.
All pointers are integers in `[1..`N`+2]`. A pointer can also be `null`.

Initially, `current = previous = 1`, all `R[i]` and `P[i]` are set to `null`, and all buffer elements are set to y_0^i.

During execution:
Writer:

- When τ_w is released:
  `previous := current`
  `current :=` some `j`$\in$`[1..`N`+2]` such that `free(j)`, where

$$\texttt{free(j)} \equiv (\texttt{previous}\neq\texttt{j} \wedge \forall\texttt{i}\in\texttt{[1..}N\texttt{].R[i]}\neq\texttt{j})$$

- While τ_w executes it writes to `B[current]`

Lower-priority reader τ_i:

- When τ_i is released:
  `if i`$\in$`[1..`N_1`] then R[i] := current` (link $\tau_w \to \tau_i$)
  `else R[i] := previous` (link $\tau_w \overset{-1}{\to} \tau_i$)
- While τ_i executes it reads from `B[R[i]]`
- When τ_i finishes:
  `R[i] := null`

Higher-priority reader τ_i':

- When τ_i' is released:
  `P[i] := previous`
- While τ_i' executes it reads from `B[P[i]]`

In case of simultaneous release of writer and readers: the action for the writer is executed first. The actions for the readers can be executed in any order.

Figure 3: The protocol DBP.

According to the algorithm, the writer will maintain a buffer array `B`, a pointer array `R` of size 2, and two pointers `current` and `previous`. Also, τ_1 maintains a local pointer `P`. Note that, since $N = 2$, `B` cannot grow larger than 4 buffers. The initial values are `current=previous=1` and `R[2]=R[3]=P[1]=null`.

A sample execution of DBP is shown in Figures 5 and 6. Figure 6 shows the values of the pointers during execution. Figure 5 shows the release, begin of execution and end of execution events for each task. Task τ_1 is released at times $0, 1, 2, 3, 4, 5$, task τ_w is released at times $0, 2, 4$, and so on. We use the notation $\tau_1, \tau_1', ...$ to denote different instances of the same task. Notice that a task instance may be "split" because of preemption: this is, for instance, the case of τ_2 which is split between the first and second cycle. The heights of the task "boxes" in the figure denote the relative priorities of the tasks.

Figure 5 also shows exactly where each task reads from and writes to at any given time (dashed and solid arrows respectively). The "boxes" at the bottom of the figure correspond to the buffer array `B` and the values stored in each buffer: y_0 is the initial (default) value, y_1 is the value written by the first instance of τ_w, and so on. Notice that `B` grows to 3 buffers in this example.

It can be verified that the synchronous semantics are preserved. For example, the first instance of reader 2, τ_2, reads the value produced by the first instance of the writer, which was released at the same time. The third instance of reader 1, τ_1'', reads the same value: this is because a unit delay is present in this case. It is worth noting that the unique instance of reader 3 shown in the figure, although it is preempted multiple times, consistently reads the correct value, namely y_1. The fact that this instance has not terminated execution when the writer is released at time 4 is what triggers the allocation of a new buffer `B[3]`.

4.2 Application of DBP to general task graphs

Applying DBP to a general task graph is easy: we con-

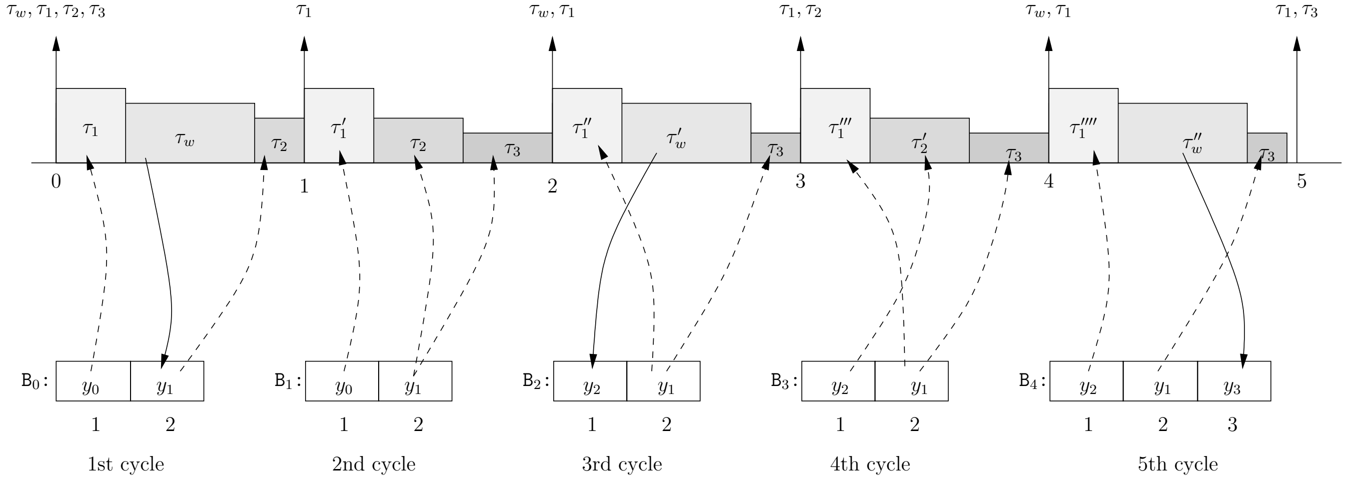

Figure 5: The execution of the tasks.

	init	0	1	2	3	4	5
`current`	1	2	2	1	1	3	3
`previous`	1	1	1	2	2	2	2
`P[1]`	null	1	1	2	2	1	1
`R[2]`	null	2	2	null	1	null	null
`R[3]`	null	2	2	2	2	2	3

Figure 6: The values of the DBP pointers during execution.

sider each writer task in the graph and apply DBP to this writer and all its readers. Of course, it is possible that a given task is both a writer and a reader: this task will subsequently appear in multiple instances of DBP in different roles. Notice that the memory spaces of the different instances of DBP (i.e., pointers, buffers, etc.) are completely separate. Thus, there is one buffer array for the first writer and its readers, another buffer array for the second writer and its readers, and so on.[6] In case of simultaneous release of many tasks, it suffices to respect the order of execution of the DBP release actions for each writer: the writer release action must be executed before the reader release actions. Notice that the writer or reader release actions associated with a given task are totally independent, since they operate on different memory spaces. Thus they can be executed in any order.

As an example, let us consider the task graph shown in Figure 7. There are three writers in this graph, namely, τ_1, τ_3 and τ_4. The buffer requirements for each of these tasks are as follows:

- τ_1 has only one lower-priority reader without unit-delay. That is, we are in the case $M = N_2 = 0$ and $N_1 = 1$. As said above, in this case DBP specializes to the high-to-low protocol, which requires one double buffer.

[6]The above apply to the case where the writer sends the same data to all readers. If the writer sends different data to different readers, then for each data and the corresponding set of readers there will be a separate instance of DBP.

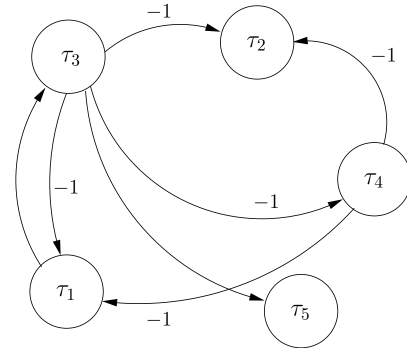

Figure 7: A task graph.

- τ_3 has two higher-priority readers τ_1 and τ_2 (with unit-delay), one lower-priority reader τ_4 without unit-delay and one lower-priority reader τ_5 with unit-delay. That is, we are in the case $N_1 = N_2 = 1$ and $M = 2$. We apply DBP and we need $N + 2 = 4$ buffers.

- τ_4 has two higher-priority readers. That is, we are in the case $N = 0$ and $M = 2$. We apply DBP and we need 2 buffers.

Thus, in total, we have 8 single buffers.

5. BUFFER REQUIREMENTS

In this section we study the buffer requirements of semantics-preserving implementations. First, we provide lower bounds on the number of buffers required in the worst case, that is, the maximum number of buffers required for any possible arrival/execution pattern. These lower bounds are equal to the number of buffers used by DBP, thus, the corresponding numbers of buffers are both necessary and sufficient. Second, we show that DBP is using buffers optimally not just in the worst case (i.e., worst arrival pattern) but in any arrival pattern.

5.1 Lower bounds on buffer requirements and optimality of DBP in the worst case

We consider again the setting of Figure 2: one writer, N_1 lower-priority readers without unit-delay, N_2 lower-priority readers with unit-delay, and M higher-priority readers (with unit-delay). Again, we let $N = N_1 + N_2$.

First, consider the case $M = N_2 = 0$ (i.e., there is no unit-delay). We claim that $N + 1 = N_1 + 1$ buffers are required in the worst case. Consider the scenario shown in Figure 8. There are $N + 1$ arrivals of the writer and one arrival of each reader. We assume that when the $(N+1)$-th arrival of the writer occurs, none of the readers has finished execution. Note that this does not violate the schedulability assumption: indeed, the readers have lower priority than the writer and may be preempted with every new release of it. In the figure we show the *lifetime* of each buffer: for `i`$= 1, ..., N$, buffer `B[i]` is used from the moment of the `i`-th arrival of the writer until the $(N+1)$-th arrival. A buffer is needed at the last arrival so that the writer does not corrupt the data stored in one of the other buffers.

Next, consider the case $M > 0$ and $N_2 = 0$ (i.e., there is a unit-delay). Then, $N_1 + 2$ buffers are required in the worst case. This can be shown using a slight modification of the previous scenario, by adding one more occurrence of the writer at the end: this is shown in Figure 9. The last buffer B$[N + 2]$ is needed because none of the first $N + 1$ buffers can be used: buffers B$[1..N]$ are used by the N lower-priority readers and buffer B$[N + 1]$ stores the previous value which may be needed when a higher-priority reader with unit-delay arrives (the latter is not shown in the figure).

Finally, consider the case $M = 0$ and $N_2 > 0$ (i.e., there is again a unit-delay). Then, $N + 2$ buffers are required in the worst case, where $N = N_1 + N_2$. A worst-case scenario is shown in Figure 10. In the first part of this scenario N_1 lower-priority readers without unit-delay arrive, interlaced with N_1 occurrences of the writer. This requires N_1 buffers. Next, N_2 lower-priority readers with unit-delay arrive, interlaced with $N_2 + 1$ occurrences of the writer as shown in the figure. This requires $N_2 + 1$ buffers since the previous values are used by the readers: reader r'_1 uses $B[N_1 + 1]$, ..., and reader r'_{N_2} uses $B[N_1 + N_2]$. The last writer cannot overwrite any of the first $N_1 + N_2$ buffers since they are used by readers that have not yet finished. The last writer cannot over-write buffer $B[N_1 + N_2 + 1]$ either, since this stores the previous value which may be needed when a lower-priority reader with unit-delay arrives (the latter is not shown in the figure).

These lower bounds show that DBP is optimal in the worst case, that is, in the "worst" arrival/execution pattern. In Section 5.3 we show that DBP actually has a stronger optimality property, in particular, it uses buffers optimally in any arrival/execution pattern.

5.2 Lower bounds in the multi-periodic case

The lower bounds provided above are for general arrival patterns. In practice, the case of *multi-periodic* arrival patterns is common, where each task is released with a fixed, known period. The lower bounds for multi-periodic arrivals are not much different from the bounds in the general case. In particular, as already observed in [19], even when the periods are powers of two, for N readers, N buffers are needed in the worst case. For instance, consider the situation where the writer has period 1 and the readers have periods $2, 4, ..., 2^N$. Then, it can be seen that at time $t = 2^N - 2$ all N buffers are needed. In particular, reader N needs the value written at time 0, reader $N-1$ needs the value written at time 2^{N-1}, reader $N - 2$ needs the value written at time $2^{N-1} + 2^{N-2}$, and so on, until reader 1 that needs the value written at time $2^{N-1} + 2^{N-2} + \cdots + 2 = 2^N - 2$. These are N different values.

5.3 Optimality of DBP for every arrival/execution pattern

The protocol DBP is in fact optimal not only in the worst case, but for every arrival/execution pattern, in the following sense:

> for every task graph, for every arrival/execution pattern of the tasks, and at any time t, the values memorized by DBP at time t are precisely those values necessary in order to preserve the semantics.

We proceed into formalizing and proving this result.

Let ρ be an arrival/execution pattern: ρ is a sequence of release, begin and end events in real-time (i.e., we know the times of occurrence of each event). We will assume that all writer tasks occur at least once in ρ, at time 0, and output their respective default values. This is simply a convention which simplifies the proofs that follow.

For an arrival/execution pattern ρ and for some time t, we define `needed`(ρ, t) to be the set of all outputs of writer tasks occurring in ρ that are still needed at time t. Formally, `needed`(ρ, t) is defined to be the set of all y such that y is the output of some writer task τ_w occurring in ρ at some time t_w, and one of the following two conditions holds:

1. There exists a link $\tau_w \to \tau_i$, task τ_i is released in ρ at the same time or after t_w and before the next occurrence of τ_w (if it exists), and τ_i finishes after t.

2. There exists a link $\tau_w \stackrel{-1}{\to} \tau_i$, there is a second occurrence of τ_w in ρ at time t'_w, where $t_w < t'_w$, τ_i is released at or after t'_w and before the next occurrence of τ_w (if it exists), and τ_i finishes after t.

We assume that outputs y are indexed by the writer identifier and occurrence number, so that no two outputs are equal and `needed`(ρ, t) contains all values that have been written. This is not a restricting assumption since in the general case the domain of output values will be infinite, thus, there is always a scenario where all outputs are different.

`needed`(ρ, t) captures precisely the minimal set of values that must be memorized by any protocol so that semantics are preserved. Another way of looking at the definitions above is that `needed`(ρ, t) contains all outputs whose *lifetime* extends from some point before t to some point after t. Notice that `needed`(ρ, t) is *clairvoyant* in the sense that it can "see" in the future, after time t. For instance, `needed`(ρ, t) "knows" whether a reader τ_i will occur after time t or not, and if so, whether this will be before the next occurrence of τ_w.

Obviously, a real implementation cannot be clairvoyant, unless it has some knowledge of the arrival/execution pattern. This motivates us to define another set of outputs that contains all outputs that *may be needed*, given the knowledge up to time t. This set is denoted `maybeneeded`(ρ, t) and

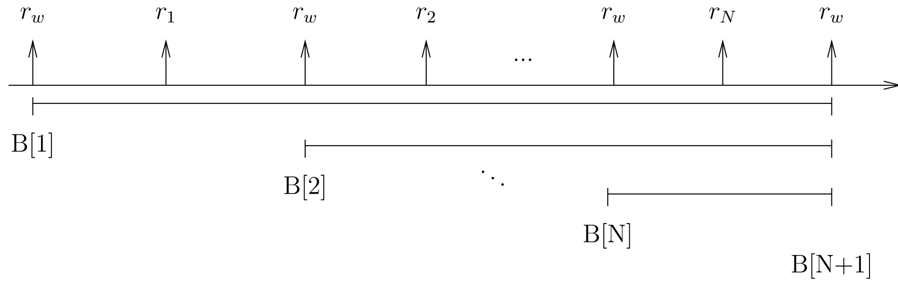

Figure 8: Worst-case scenario for $N+1$ buffers: N lower-priority readers without unit-delay.

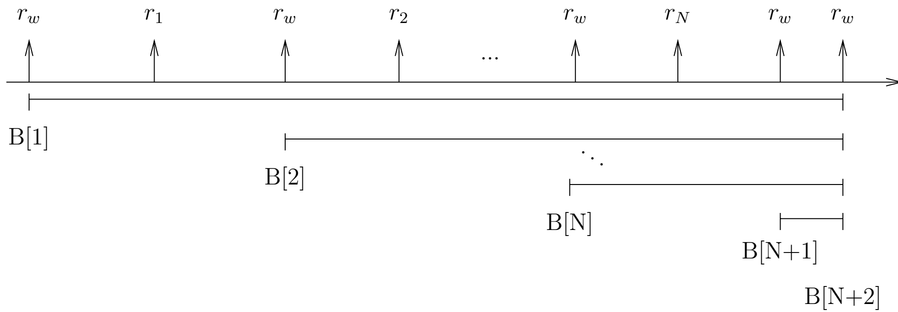

Figure 9: First worst-case scenario for $N+2$ buffers: N lower-priority readers without unit-delay and at least one higher-priority reader (with unit-delay).

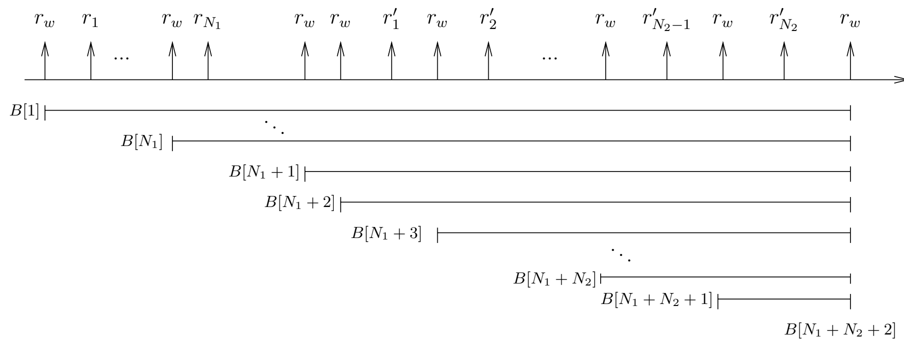

Figure 10: Second worst-case scenario for $N+2$ buffers: $N = N_1 + N_2$, N_1 lower-priority readers without unit-delay and N_2 lower-priority readers with unit-delay.

it is formally defined to be the set of all y such that y is the output of some writer task τ_w occurring in ρ at some time t_w, and one of the following two conditions holds:

1. There exists a link $\tau_w \rightarrow \tau_i$, such that, if there is a second occurrence of τ_w in ρ at time t'_w, with $t_w < t'_w < t$, then there is an occurrence of τ_i at time t_i, with $t_w \leq t_i < t'_w$, and τ_i finishes after t.

2. There exists a link $\tau_w \stackrel{-1}{\rightarrow} \tau_i$, such that, if there is a second and a third occurrence of τ_w in ρ at times t'_w and t''_w, with $t_w < t'_w < t''_w < t$, then there is an occurrence of τ_i at time t_i, with $t'_w \leq t_i < t''_w$, and τ_i finishes after t.

The intuition is that y may be needed because the reader task τ_i may perform a read operation, say, right after time t. It should be clear that for any ρ and t, $\texttt{needed}(\rho, t) \subseteq \texttt{maybeneeded}(\rho, t)$.

We want to compare the values stored by DBP to the above sets. To this end, we define $\texttt{DBPused}(\rho, t)$ as the set of all values stored in some buffer `B[i]` of DBP at time t, when DBP is executed on the arrival/execution pattern ρ, such that `free(i)` is false[7] (recall that the predicate `free` is defined in Figure 3).

We then have the following result.

Theorem 1. *For any arrival/execution pattern ρ and any time t,*

$$\texttt{DBPused}(\rho, t) \subseteq \texttt{maybeneeded}(\rho, t).$$

Proof. Consider some y in $\texttt{DBPused}(\rho, t)$. There must be some position `j` such that `free(j)` is false and the value of `B[j]` at time t is y. This value was written by the writer τ_w at time $t_w < t$. We reason by cases:

1. `free(j)` is false because `previous`=`j`. This means that there is a reader task τ_i communicating with τ_w with a unit-delay link $\tau_w \stackrel{-1}{\rightarrow} \tau_i$. We must show that Condition 2 in the definition of $\texttt{maybeneeded}(\rho, t)$ holds. We consider the following cases, depending on how many times τ_w was released before t:

    - τ_w is not released before t. This means that `previous` = `j` = 1 and `B[j]` holds the default value y_0.

    - τ_w is released only once before t. When this happens, `previous` is set to `current` which is equal to 1, since this is the first release of τ_w. Thus, again `B[j]` holds the default value y_0.

    - τ_w is released at least twice before t, and the last two times where at $t_w < t'_w < t$. At t'_w, `previous` is set to `current` which equals `j` at that point. Thus, `B[j]` holds the value y written by the instance of τ_w released at t_w.

    In none of the three cases above there is more than one occurrence of τ_w after t_w, thus Condition 2 in the definition of $\texttt{maybeneeded}(\rho, t)$ holds.

2. `free(j)` is false because there is some $i \in [1..N_1]$ such that `R[i]`=`j`. This means that the reader τ_i communicates with τ_w via a link without unit-delay, $\tau_w \rightarrow \tau_i$. Since `R[i]`$\neq$`null`, τ_i is released at least once before t and it has not finished at time t. Suppose τ_i is released last at time $t_i < t$. When this happens, `R[i]` is set to `current` which equals `j` at that point. Condition 1 in the definition of $\texttt{maybeneeded}(\rho, t)$ holds since t_w is the last occurrence (if any) of the writer before time t_i and τ_i finishes after time t.

3. `free(j)` is false because there is some `i`$\in [N_1+1..N_1+N_2]$ such that `R[i]`=`j`. This means that the reader τ_i communicates with τ_w via a link with unit-delay, $\tau_w \stackrel{-1}{\rightarrow} \tau_i$. This case is similar to Case 1 above.

□

The above result shows that DBP never stores redundant data, only data that may be needed. In the absence of any knowledge about the future (which is unknown if arrival/execution patterns are not known), this is the best that can be achieved, if we are to preserve semantics.

6. CONCLUSIONS AND PERSPECTIVES

We have studied the problem of semantics-preserving implementations of synchronous (zero-time) semantics. We have proposed a general protocol called DBP, that implements a non-blocking, buffer-based, inter-task communication scheme that preserves semantics for any arrival pattern of the tasks. We have also shown that DBP is optimal in terms of buffer usage, both in the worst case but also for every arrival/execution pattern, assuming no knowledge about future arrivals.

In the case where the arrival pattern of tasks is known a-priori, for instance, when tasks are multi-periodic, DBP can be easily turned into a static protocol. This can be done by *simulating* the behavior of DBP on the given arrival pattern and determining the buffers in advance. In the multi-periodic case this simulation must be performed up to the *hyper-period* of the tasks, that is, the least common multiple of all periods. Although DBP, as presented above, will not be optimal in this case, simple modifications suffice to regain optimality. In particular, it suffices to check, upon arrival of the writer, whether this output will be needed in the future. Since future arrivals of readers are known, this test is feasible. If the output is not needed, then no buffer needs to be used. The details are left for the full version of this paper.

Future objectives include relaxing the schedulability assumption, for instance, to cases where the deadline of a task is allowed to be greater than its minimum inter-arrival time. This implies that when a new instance of a task arrives the previous instance may not have necessarily finished. We suspect that a protocol similar to DBP can be devised in this case, at the expense of additional buffers.

We also plan to extend this work to distributed, multiprocessor execution platforms. This has been partly done in [6] for synchronous distributed architectures and multiperiodic tasks, where static, non-preemptive scheduling was

[7] When implementing DBP, there is the option of *pre-allocating* the worst-case number of buffers or allocating buffers *on-the-fly*, that is, during execution, as necessary. This is a usual time vs. space trade-off. To avoid such implementation considerations, we have included in the above definition of $\texttt{DBPused}(\rho, t)$ the requirement that `free(i)` be false, which means that, even if `B[i]` has been pre-allocated, its contents are not needed anymore.

assumed. We still need to cover "loosely" synchronous [4] or asynchronous architectures with preemptive scheduling and more general task arrival patterns.

7. REFERENCES

[1] Audsley, N., Burns, A., Davis, R., Tindell, K., and Wellings, A. Fixed priority pre-emptive scheduling: An historical perspective. *Real Time Systems 8*, 2-3 (1995).

[2] Baleani, M., Ferrari, A., Mangeruca, L., and Sangiovanni-Vincentelli, A. Efficient embedded software design with synchronous models. In *5th ACM International Conference on Embedded Software (EMSOFT'05)* (2005), pp. 187 – 190.

[3] Benveniste, A., and Berry, G. The synchronous approach to reactive and real-time systems. *IEEE Proceedings 79* (Sept. 1991), 1270–1282.

[4] Benveniste, A., Caspi, P., Guernic, P. L., Marchand, H., Talpin, J., and Tripakis, S. A protocol for loosely time-triggered architectures. In *Embedded Software (EMSOFT'02)* (2002), vol. 2491 of *LNCS*, Springer.

[5] Bhattacharyya, S. S., Murthy, P. K., and Lee, E. A. *Software Synthesis from Dataflow Graphs*. Kluwer, 1996.

[6] Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)* (2003), ACM.

[7] Chen, J., and Burns, A. A three-slot asynchronous reader-writer mechanism for multiprocessor real-time systems. Tech. Rep. YCS-286, Department of Computer Science, University of York, May 1997.

[8] Fersman, E., and Yi, W. A generic approach to schedulability analysis of real-time tasks. *Nordic J. of Computing 11*, 2 (2004), 129–147.

[9] Halbwachs, N. *Synchronous Programming of Reactive Systems*. Kluwer, 1992.

[10] Harbour, M., Klein, M., Obenza, R., Pollak, B., and Ralya, T. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Kluwer, 1993.

[11] Huang, H., Pillai, P., and Shin, K. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX'02* (2002).

[12] Lee, E., and Messerschmitt, D. Synchronous data flow. *Proceedings of the IEEE 75*, 9 (1987), 1235–1245.

[13] Liu, C., and Layland, J. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (Jan. 1973), 46–61.

[14] Murthy, P. K., and Bhattacharyya, S. S. Buffer merging — a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems 9*, 2 (April 2004), 212–237.

[15] Natale, M. D. Optimizing the multitask implementation of multirate Simulink models. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)* (2006). To appear.

[16] Scaife, N., and Caspi, P. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)* (2004).

[17] Sha, L., Rajkumar, R., and Lehoczky, J. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers* (Sept. 1990).

[18] Stankovic, J., Spuri, M., Ramamritham, K., and Buttazzo, G. *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[19] Tripakis, S., Sofronis, C., Scaife, N., and Caspi, P. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or EDF schedulers. In *5th ACM International Conference on Embedded Software (EMSOFT'05)* (2005), pp. 353 – 360.

APPENDIX

A. PROOF OF CORRECTNESS OF THE DYNAMIC BUFFERING PROTOCOL

What we want to prove is semantical preservation, that is, that for any possible arrival pattern and values written by the writer, the values read by the readers in the ideal semantics are equal to the values read by the readers in the implementation, assuming DBP is used. More formally, consider a reader τ_i and let t_i be the time when an arbitrary instance of τ_i is released. We denote this instance by $\tau_i^{t_i}$. Let $t_i' \geq t_i$ be the time when $\tau_i^{t_i}$ reads. Let τ_w be the writer task. For the moment, let us assume that τ_w is released at least twice before time t_i. We relax this assumption later in this section.

Let $t \leq t_i$ be the last time before t_i that an instance of τ_w was released. We denote this instance by τ_w^t. Let $t_e > t$ be the time that τ_w^t produces its output and finishes. Let $y(t)$ be the output of τ_w^t. Let $t' < t$ be the last time before t that an instance of the writer τ_w was released. This instance is denoted $\tau_w^{t'}$. It finishes execution at time $t_e' > t'$. Let $y(t')$ be the output of $\tau_w^{t'}$. Figure 11 illustrates the notation defined above. Notice that the order of events shown in the figure is just one of the possible orders.

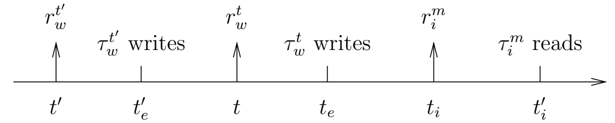

Figure 11: Illustration used in the proof of DBP.

A.1 Lower-priority reader without unit-delay

Suppose, as a first case, that the reader τ_i has a lower priority than the writer τ_w and we have $\tau_w \to \tau_i$. Let $x(t_i)$ be the value read by $\tau_i^{t_i}$. The ideal semantics states that $x(t_i) = y(t)$. We want to show that this equality holds in the implementation as well.

Let us first handle the case where the writer is never released before time t_i. In this case, $y(t)$ is equal to the default output of τ_w. Also, when $\tau_i^{t_i}$ is released, `R[i]` is set to 1, which is the initial value of `current` (Figure 3). `R[i]` is not modified in the interval $[t_i, t_i']$. Thus, at time t_i', τ_i reads the value stored in buffer `B[1]`. This is the default output of τ_w, since no buffer has been written by the writer yet.

Let us now turn to the case where the writer is released at $t \leq t_i$. Recall that the writer chooses upon release a "free" position in the buffer array where it will write to (Figure 3). Such a free position always exists by the pigeon-hole principle, as already mentioned. Let j^t be the position that τ_w^t chooses. Let R^t, p^t and c^t be the values of `R`, `previous` and `current` at time t, right after the execution of the assignments `previous := current` and `current := ...`. Then, by definition of DBP, the following hold:

$$p^t \neq j^t \text{ and } \forall i \in [1..n].R^t[i] \neq j^t \text{ and } c^t = j^t.$$

$t_e \geq t$ is the time when τ_w^t finishes writing: let B^{t_e} be the value of `B` after this write operation.[8] Then, since `current`

[8]Notice that in Figure 11 we have $t_e < t_i$ but this need not be the case. We could also have $t_e > t_i$.

is not modified between t and t_e and $c^t = j^t$ (this is because `current` can only be changed by a newer release of the same writer), we also have:

$$B^{t_e}[j^t] = y(t).$$

Now consider the reader $\tau_i^{t_i}$. Again, `current` is not modified between t and t_i, thus, we have:

$$R^{t_i}[i] = c^t = j^t.$$

$\tau_i^{t_i}$ reads the value

$$B^{t'_i}[R^{t'_i}[i]] = B^{t'_i}[R^{t_i}[i]] = B^{t'_i}[j^t].$$

This is because `R[i]` is not modified between t_i and t'_i.

To show that $\tau_i^{t_i}$ read the correct value $y(t)$, we must show that $B^{t'_i}[j^t] = B^{t_e}[j^t]$, that is, that the position j^t is not over-written between t_e and t'_i. This is because only the writer can write into `B[`j^t`]` and in order to do so it must choose j^t as a free position. Since the writer does not arrive in the interval $[t_e, t_i]$, it suffices to show that `free(`j^t`)` is false in the interval $[t_i, t'_i]$. This is because `R[i]` equals j^t in all this interval.

A.2 Lower-priority reader with unit-delay

Suppose, next, that the reader τ_i has a lower priority than the writer τ_w and we have a link with a unit-delay: $\tau_w \stackrel{-1}{\rightarrow} \tau_i$. Again, let $x(t_i)$ be the value read by $\tau_i^{t_i}$. The ideal semantics states that $x(t_i) = y(t')$. We want to show that this equality holds in the implementation as well.

Let us first handle the case where the writer is released not more than once before time t_i. In this case, $y(t')$ is equal to the default output of τ_w. Also, when $\tau_i^{t_i}$ is released, `R[i]` is set to 1, which is the value of `previous` at this point. Indeed, either the writer has never been released yet and `previous` is equal to its initial value 1, or the writer has been released once and `previous` is set to the initial value of `current`, which is also 1. `R[i]` is not modified in the interval $[t_i, t'_i]$. Thus, at time t'_i, τ_i reads the value stored in buffer `B[1]`. If the writer has not been released before t_i then `B[1]` holds the default output of τ_w. If the writer has been released once before t_i then it has not written to `B[1]`: to do so, it must choose 1 as a free position to assign to `current`, however, 1 is not free because `previous=1`.

Let us now turn to the case where the writer is released twice before t_i. Upon arrival of the writer at time t', a free position in the buffer array is chosen to write to: let this position be $j^{t'}$. Let also $R^{t'}, p^{t'}$ and $c^{t'}$ be the values of `R`, `previous` and `current` at time t', right after the execution of the assignments to `previous` and `current`. Then, by definition of DBP, the following hold:

$$p^{t'} \neq j^{t'} \text{ and } \forall i \in [1..n].R^{t'}[i] \neq j^{t'} \text{ and } c^{t'} = j^{t'}.$$

$t'_e \geq t'$ is the time when $\tau_w^{t'}$ finishes writing: let $B^{t'_e}$ be the value of `B` after this write operation. Then, since `current` is not modified between t' and t'_e and $c^{t'} = j^{t'}$, we also have:

$$B^{t'_e}[j^{t'}] = y(t').$$

On the next arrival of the writer at time t, new assignments will be made to the pointers `previous` and `current`. Let j^t be the new free position chosen. Let R^t, p^t and c^t be the values of `R`, `previous` and `current` at time t, right after the assignments to `previous` and `current`. Then, the following hold:

$$p^t = c^{t'} \text{ and } p^t \neq j^t \text{ and } \forall i \in [1..n].R^t[i] \neq j^t \text{ and } c^t = j^t$$

When the reader arrives at time t_i, `R[i]` is set to `previous`. `previous` is not modified between t and t_i, thus, the value of `R[i]` at time t_i is equal to p^t:

$$R^{t_i}[i] = p^t = c^{t'} = j^{t'}.$$

`R[i]` is not modified between t_i and t'_i. Thus, $\tau_i^{t_i}$ reads the value

$$B^{t'_i}[R^{t'_i}[i]] = B^{t'_i}[R^{t_i}[i]] = B^{t'_i}[j^{t'}].$$

To show that $\tau_i^{t_i}$ reads the correct value $y(t')$, we must show that $B^{t'_i}[j^{t'}] = B^{t'_e}[j^{t'}]$, that is, that the position $j^{t'}$ is not over-written between t'_e and t'_i. This is because only the writer can write into `B[`$j^{t'}$`]` and in order to do so it must choose $j^{t'}$ as a free position. Since the writer does not arrive in the interval $[t'_e, t]$, it suffices to show that `free(`$j^{t'}$`)` is false in the interval $[t, t'_i]$. In the interval $[t, t_i]$, `free(`$j^{t'}$`)` is false because `previous` equals $j^{t'}$. In the interval $[t_i, t'_i]$, `free(`$j^{t'}$`)` is false because `R[i]` equals $j^{t'}$.

A.3 Higher-priority reader (with unit-delay)

Now consider the case where $\tau_i^{t_i}$ is a higher-priority task. Thus, the link is $\tau_w \stackrel{-1}{\rightarrow} \tau_i$. Let again $x(t_i)$ be the value read by $\tau_i^{t_i}$. We must show that $x(t_i) = y(t')$.

The case where the writer is released not more than once before time t_i is identical to the corresponding case in Section A.2. We thus omit it and turn directly to the case where the writer is released twice before t_i. Let $c^{t'}$ be the value of `current` that is chosen at time t'. Since `current` is not modified between t' and t, we have:

$$p^t = c^{t'}.$$

The value $y(t')$ is written in buffer position $c^{t'}$ and this is not modified until t, when the writer is released next. At this point, $p^t \neq j^t$, or $c^{t'} \neq j^t$, thus, this position is not over-written by the instance τ_w^t.

`previous` is not modified between t and t_i, thus, we have:

$$P^{t_i}[i] = p^t = c^{t'}.$$

`P[i]` is not modified between t_i and t'_i, thus, at time t'_i, $\tau_i^{t_i}$ reads the value

$$B^{t'_i}[P^{t'_i}[i]] = B^{t'_i}[P^{t_i}[i]] = B^{t'_i}[p^t] = B^{t'_i}[c^{t'}].$$

To show that $\tau_i^{t_i}$ reads the correct value $y(t')$, we must show that the position $c^{t'}$ is not over-written by any instance of the writer until time t'_i. This is true because in order for the writer to write in position $c^{t'}$, it must choose it as a free position. Note that no instance of the writer is released between t and t_i, by definition of t and t_i. Also, if an instance of the writer is released between t_i and t'_i, this instance cannot execute before $\tau_i^{t_i}$ finishes, because it has lower priority than $\tau_i^{t_i}$.