BAMBERGER BEITRÄGE

ZUR WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK

ISSN 0937-3349

Nr. 103

Optimized Buffering of Time-Triggered Automotive Software

Eugene Yip • Erjola Lalo • Gerald Lüttgen • Michael Deubzer • Andreas Sailer

September 2018

URN: urn:nbn:de:bvb:473-opus4-529175
DOI: https://doi.org/10.20378/irbo-52917

FAKULTÄT WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK

OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG

Lehrstuhl Softwaretechnik & Programmiersprachen
Fakultät Wirtschaftsinformatik & Angewandte Informatik

Optimized Buffering for Time-Triggered Automotive Software*

Eugene Yip,[1] Erjola Lalo,[2] Gerald Lüttgen,[1] Michael Deubzer,[2] Andreas Sailer[2]

[1]Otto-Friedrich-Universität Bamberg, 96045 Bamberg, Germany
[2]Vector Informatik GmbH, Franz-Mayer-Str. 1, 93053 Regensburg, Germany

September 21, 2018

Abstract: The development of an automotive system involves the integration of many real-time software functionalities, and it is of utmost importance to guarantee strict timing requirements. However, the recent trend towards multi-core architectures poses significant challenges for the timely transfer of signals between processor cores so as to not violate data consistency.

We have studied and adapted an existing buffering mechanism to work specifically for statically scheduled time-triggered systems, called *static buffering protocol*. We developed further buffering optimisation algorithms and heuristics, to reduce the memory consumption, processor utilisation, and end-to-end response times of time-triggered AUTOSAR designs on multi-core platforms. Our contributions are important because they enable deterministic time-triggered implementations to become competitive alternatives to their inherently non-deterministic event-triggered counterparts. We have prototyped a selection of optimisations in an industrial tool and evaluated them on realistic industrial automotive benchmarks.

*Research support was provided by the Bayerische Forschungsstiftung under grant no. AZ-1257-16, project OBZAS.

Contents

1 Introduction

The development of an automotive system involves the integration [OSHK09] of many real-time software functionalities, where it is critical to guarantee strict timing requirements. The *automotive open system architecture* (AUTOSAR) standard [AUT17a] is popular for developing modular software components with high interoperability. An important type of requirement, called *end-to-end response time*, specifies the maximum time that the system can take to deliver an output to a corresponding input. Such timing requirements are easier to guarantee with time-triggered implementations because they offer better time-predictability than their event-triggered counterparts [Kop91]. However, the recent trend towards multi-core architectures poses significant challenges for the timely transfer of data and control signals between processor cores so as not to violate data consistency.

In this light, the automotive industry [EKQS18, HvHM+16, RNH+15] has shown great interest in using the *logical execution time* (LET) task model [KS12] for designing time-triggered multi-core systems. A LET task has a statically defined period and block of time, called the *logical execution time*, during which the task is allowed to execute its computations. Task communication via signals is limited to the start and end of each LET, and is idealised to complete in zero time. This ensures time-predictable and deterministic communication that is unaffected by changes in the underlying platform [HK07]. This platform invariant property is attractive to automotive manufacturers as it greatly simplifies the migration of legacy single-core software to multi-core platforms [HvHM+16, RNH+15]. The automotive industry is also taking advantage of LET tasks as design contracts between control and software engineers, and between component suppliers and system integrators [EKQS18]. However, signal buffering is needed to preserve the data- and control-flow between the tasks [FNG+09], especially when their LETs do not align. Thus, significant time may be spent on managing the buffers, and significant memory may be needed for the buffers [FNG+09].

1.1 Contributions

Despite the increasing interest in the LET time-triggered approach, event-triggered systems remain popular because of their ability to achieve better average-case response times and resource utilisation [Kop91]. To improve the practicality of the time-triggered approach, we present an adaptation of the *dynamic buffering protocol* (DBP) [STC06] that is suitable for LET communication, and develop buffering optimisation algorithms and heuristics to reduce the memory consumption, processor utilisation, and end-to-end response times for multi-core time-triggered AUTOSAR designs. The algorithms and heuristics synthesise the required buffers and associated accesses for each signal, and the mapping of tasks to processor cores. When adapting existing buffering protocols to LET tasks, attention is needed to the fact that LET communication is defined to occur instantaneously at predefined time points. Our contributions are important to allow time-triggered implementations to become competitive alternatives to their event-triggered counterparts.

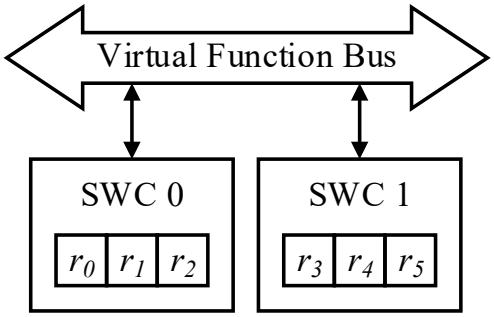
1.2 Report structure and Content

Section 2 recalls (1) the AUTOSAR methodology for developing automotive systems, (2) the scheduling of LET tasks, and (3) the challenges with implementing a system that preserves the LET semantics. Section 3 discusses related work on buffering protocols developed for real-time task communication. We find that DBP is a good candidate for buffering LET communication. Section 4 presents related work on algorithms and heuristics developed for

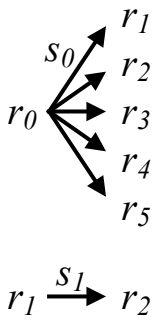
optimising AUTOSAR designs, focussing on the execution time and memory cost of task communication and on end-to-end response times.

Section 5 discusses the heterogeneous hardware and software architecture that is assumed, followed by an overview of our proposed buffering optimisation approach in Section 6. Our approach consists of optimisations that are applied during the design and deployment of an LET system. The overall optimisation goal is to reduce processor and memory utilisation due to task communication, and to reduce end-to-end response times. The design phase optimisations (see Section 7) include the adaptation of DBP to statically scheduled LET tasks (called *static buffering protocol*, SBP), and the suppression of unnecessary signal writes. Our optimisations support signals to which multiple task write, and signals that may be assigned several values before stabilising on a final value. The deployment phase optimisations (see Section 8) formulate the assignment of signal buffers-to-memory modules, of tasks-to-cores, and of buffering protocols to each signal as a mixed-integer linear programming (MILP) problem. Because solving resource allocation problems is NP-hard, a genetic algorithm of the MILP problem is provided for situations where possibly suboptimal solutions are acceptable for faster solving time. Once memory allocations are found for the signal buffers, a heuristic is used to merge buffers with disjoint lifetimes.

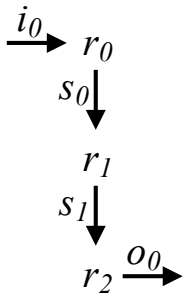
We evaluated a selection of the proposed optimisations on synthetic benchmarks, based on actual airbag, chassis, and engine management systems, and on an industrial engine management system from the FMTV Challenge [HDK+17]. Section 9 describes the implementation of the selected optimisations in the TA Tool Suite [Vec18], which aids AUTOSAR designers in modelling, designing, and analysing the timing behaviour of event-triggered or time-triggered multi-core automotive software. Sections 10 and 11 explain the setup of the synthetic and industrial benchmarks, respectively, and discuss preliminary results that suggest that LET-based AUTOSAR designs with SBP require less memory and execution time than with the traditional point-to-point communication approach. Finally, Section 12 provides concluding remarks on the optimisation of LET communication in AUTOSAR designs.



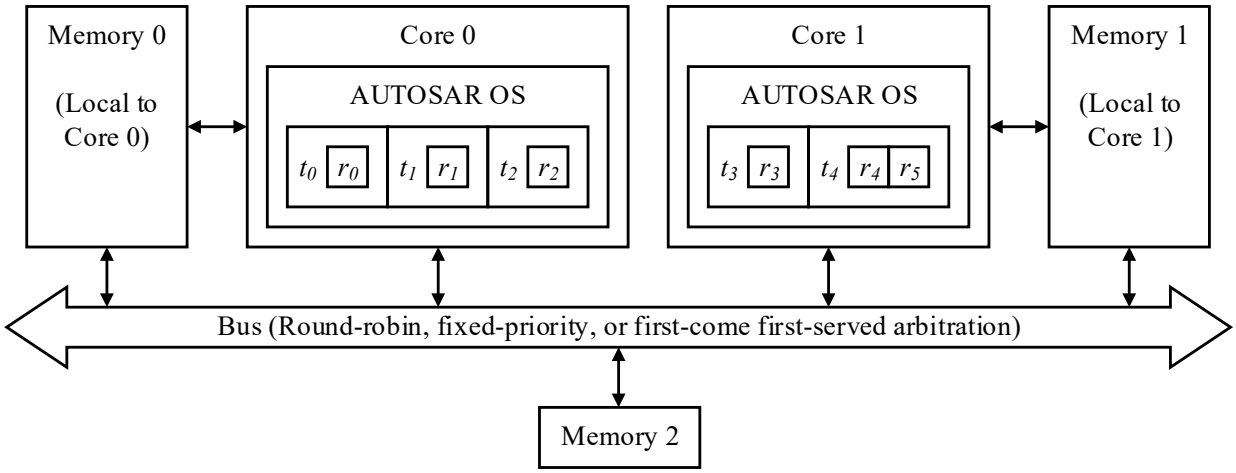
(a) Design phase (r is a runnable).



(b) Runnable communication dependencies for signals s_0 and s_1.



(c) Event-chain ec_0, with input i_0 and output o_0.



(d) Deployment phase (t is a task).

Figure 1: AUTOSAR methodology.

2 Background

This section discusses the challenges surrounding the deployment of AUTOSAR designs onto multi-core platforms. Of note is the need to ensure data consistency among communicating tasks, and the desire to maintain time-predictable behaviour among the possible platform configurations.

2.1 AUTOSAR Methodology

An AUTOSAR design [AUT17a] consists of one or more self-contained software components (SWCs) that communicate over memory-mapped signals. A software component contains one or more so-called *runnables* that each encapsulate the smallest code-fragment that can be scheduled by an operating system. Figure 1a exemplifies a small design with two SWCs and six runnables. Runnables communicate over signals, and Figure 1b shows some dependencies for the signals s_0 and s_1. For signal s_0, runnable r_0 is the sole writer and runnables r_1 to r_5 are the readers. For signal s_1, runnable r_1 is the writer and runnable r_2 is the reader. Communication dependencies influence the execution order of the runnables, and cyclic dependencies are broken by delaying one of the communication links.

When deploying an AUTOSAR design, runnables are mapped to operating system tasks. Due to resource constraints, AUTOSAR-compliant operating systems typically only support a limited number of tasks and several runnables may be mapped to the same task. The

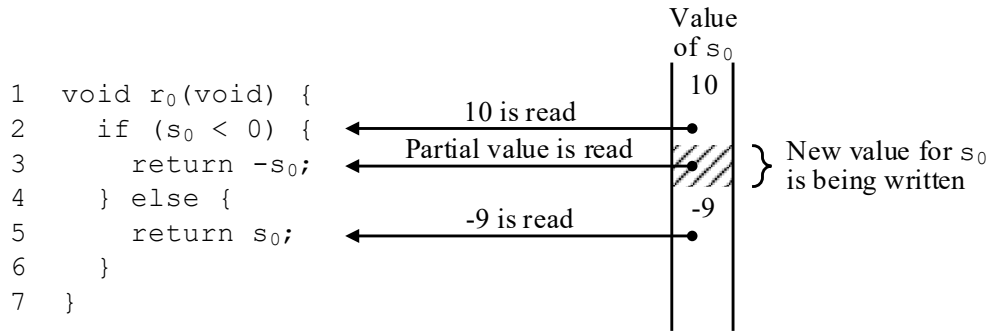


Figure 2: Example of signal stability and partial reading issues.

mapping also depends on whether a runnable contains specific computations that can only be executed or accelerated by a particular type of processor core (e.g., floating point or digital signal processing operations) or needs to access specific peripherals for sensing or actuating. In such a case, several runnables from different SWCs may need to be mapped to the same task to be executed by a specific core. Figure 1d shows a possible multi-core deployment of Figure 1a. It is common for an input signal to be processed by a sequence of runnables, and an *event-chain* [KKTM10] can be used to capture the causal relationships between event occurrences. The event-chain ec_0 of Figure 1c defines that input i_0 is processed by runnables r_0, r_1, and r_2 to produce output o_0, with intermediate signals s_0 and s_1 being produced along the way. The time that an event-chain needs to generate an output from its input is its *end-to-end response-time*. Data age constraints [AUT17c], such as $r_0 \xrightarrow{s_0,\ \delta} r_1$, can be specified to enforce that the value of s_0 read by r_1 must not have been written by r_0 more than δ time units ago.

After mapping the tasks to a multi-core platform, a scheduling discipline is selected to manage the sharing of resources (e.g., memory and processor time) among the tasks. Incorrect values may be communicated between tasks if insufficient time is given to complete the communications, or insufficient (buffer) memory is allocated. In such cases, the implementation is incorrect and must be rectified, e.g., by redesigning the software or by provisioning more resources. Static timing analysis [WEE+08] is typically performed to validate the real-time behaviour of the system before it is placed into operation.

2.2 Preemptive Task Scheduling and Data Consistency

AUTOSAR [AUT17a] defines the use of *AUTOSAR OS* as the basis for fixed-priority preemptive task scheduling [LL73] to preferentially execute higher priority tasks for shorter response times. When a higher priority task is activated, e.g., by a periodic timer, the scheduler interrupts the executing task and begins to execute the higher priority task. The scheduler saves the execution context of the preempted task so that its execution can be resumed later, after all the higher priority tasks have completed their executions. Preemption can cause non-deterministic timing behaviours, because task interruptions depend on their priorities and actual execution times. This results in end-to-end response times with high jitter, which is undesirable for real-time automotive systems.

Preemptive scheduling can cause signal writes and reads to interleave among the tasks, leading to inconsistent values being read. For example, in Figure 2, runnable r_0's code for returning the absolute value of signal s_0 is shown on the left side, and s_0's value over time is shown along the right side. The runnable begins by reading the value 10 for s_0, which is a positive number. It attempts to return the original value of s_0, which is updated to -9 in the meantime. Hence, an incorrect value is returned because s_0's value was *unstable* during r_0's

execution. If instead s_0's value is read while it is updated (e.g., line 3 in Figure 2), then only a *partial* value is read. Signal stability and partial reading issues can cause runnables to branch along incorrect paths or to compute incorrect values for other signals.

When a task needs to read from multiple signals, it is possible that some of the signals are tightly coupled, e.g., the sampling of an engine's temperature and rotational speed as two periodic signals. A task reads such tightly coupled signals in a *coherent* manner if it reads the n-th value of each signal together. In any real implementation, it takes time to deliver sensor values to the tasks. Hence, the system must be robust against delays because they can cause tasks to read different signal instances together (incoherent reads). It is the responsibility of the system designer to define the coherent signals. We only address the concerns for data stability and the prevention of partial reads by using appropriate data protection mechanisms. Signal coherency builds on top of signal stability and would require signal instances to be tracked at run-time. We consider signal coherency as future work.

2.3 Data Protection Mechanisms

Data protection mechanisms [HZN+14, Ray13, BCB+08], e.g., locks, are needed to give tasks exclusive access to signals. However, the use of locks in real-time multi-core systems is undesirable [HZN+14] because they can cause parallel tasks to block and sequentialise their executions, to suffer from deadlocks, and to experience priority inversions where higher priority tasks are blocked by lower priority tasks. Thus, locks introduce additional inter-core interferences that are complex to analyse [GGL14].

Lock-free methods [Her90] attempt to minimise the blocking time by allowing tasks to access signals without locks. An access is successful if no other task has updated the signal at the same time. Otherwise, the access must be retried until successful. The number of retries can be bounded [Her90] to estimate the worst-case access time. It should be noted that lock-free methods solve the partial read issue, but do not provide signal stability.

Wait-free methods [Her90] provide a strategy that is based on keeping snapshots of a signal's value from different points of time, and tasks access specific snapshots stored in buffers. This enables tasks to access signals independently and concurrently without having to block or retry, making wait-free methods amenable to static timing analysis. Once a snapshot is no longer needed by any task, its buffer element can be reused for a new snapshot. Since a signal's value in a snapshot is constant, signal stability can be guaranteed. Compared to locks and lock-free methods, wait-free methods provide short predictable access times and signal stability, but may require significant buffer memory to be allocated. Section 3 reviews a selection of wait-free methods developed for real-time systems.

2.4 Logical Execution Time (LET) Task Model

The LET task model [KS12] was originally developed as part of the time-triggered Giotto language [HHK01]. It is being used by the automotive industry to enhance legacy embedded control software with real-time behaviour [RNH+15] and to parallelise their execution [HvHM+16]. Figure 3 illustrates the parameters of a LET task: the *period* contains a block of time, called the logical execution time (LET), during which the task can execute its computations for up to its worst-case execution time ($WCET$). If the task fails to complete before the end of its LET, i.e., the task's deadline, then a timing error occurs and it must be handled by the run-time (e.g., by dropping the task instance). The start of the LET is determined by an *activation offset*, which can be zero. All tasks start their first period together when the system is initialised. A positive *initial task offset* can be specified to

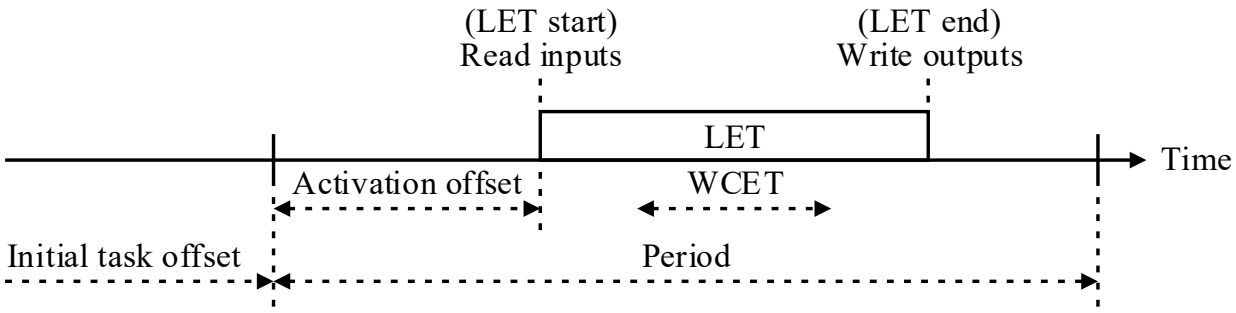


Figure 3: Parameters of a LET task.

delay the start of the task's first period. The end of a task's period coincides with the start of its next period. The following constraints can be used to validate a task's parameters:

- period $\geq$ activation offset + LET: Ensures that the period is long enough to contain the LET;
- LET $\geq$ WCET: Ensures that the LET provides enough time to execute the task's computations.

The task reads its input signals at the start of its LET and their values remain constant throughout the LET. The task writes its output signals at the end of its LET. The writing and reading of signals at the LET boundaries is idealised to occur instantaneously in zero time, thus guaranteeing by design that signal values are updated atomically and remain stable during task execution. Because task communication only occurs at the LET boundaries, the task's input/output behaviour is time-predicable and decoupled from the task's computation time. Although this greatly simplifies the static analysis of end-to-end response times, it also imposes an artificial delay on signal communication, which the implementation must preserve.

2.5 Static Scheduling of LET Tasks

AUTOSAR [AUT17a] defines the use of schedule tables (for each core) to implement time-triggered systems. A schedule table defines a sequence of task activations to be performed at predefined times, and can be constructed using the *base-period* [YKRB14] or *hyper-period* [YKRB14, CM05] approach. In the base-period approach, time is divided into equally sized slots, called the base-period, in which tasks are allocated some time to execute their computations. As a result, tasks are executed preemptively in a time-sliced manner. Its main advantage is the ability to reuse the slack that builds up at the end of each base-period, so as to support variable task periods [YKRB14]. However, scheduling overheads become significant when the base-period is much shorter than the task periods. The hyper-period approach constructs longer schedules that contain consecutive instances of each task. The hyper-period approach allows for better schedulability than the base-period approach, because computations can be scheduled over the entire task period, such that unnecessary time-slicing preemptions are avoided.

For this work, LET tasks are statically scheduled using the hyper-period approach because support for variable task periods is not needed. Figure 4 shows a 6 *ms* hyper-period schedule for the tasks in Table 1. The first step in constructing a hyper-period schedule is to allocate the WCET of each task (shaded segments in Figure 4) within the LET of their initial period. Subsequent task instances are appended to the schedule until all tasks end

Table 1: Example timing information (in ms) for the tasks in Figure 1d

Task	Period	LET	WCET	Initial Offset	Activation Offsets
t_0	1	0.5	0.25	0	0
t_1	1.2	1	0.25	0	0
t_2	1.5	1	0.5	0	0
t_3	2	1.5	1	0	0
t_4	6	4	2	0	0

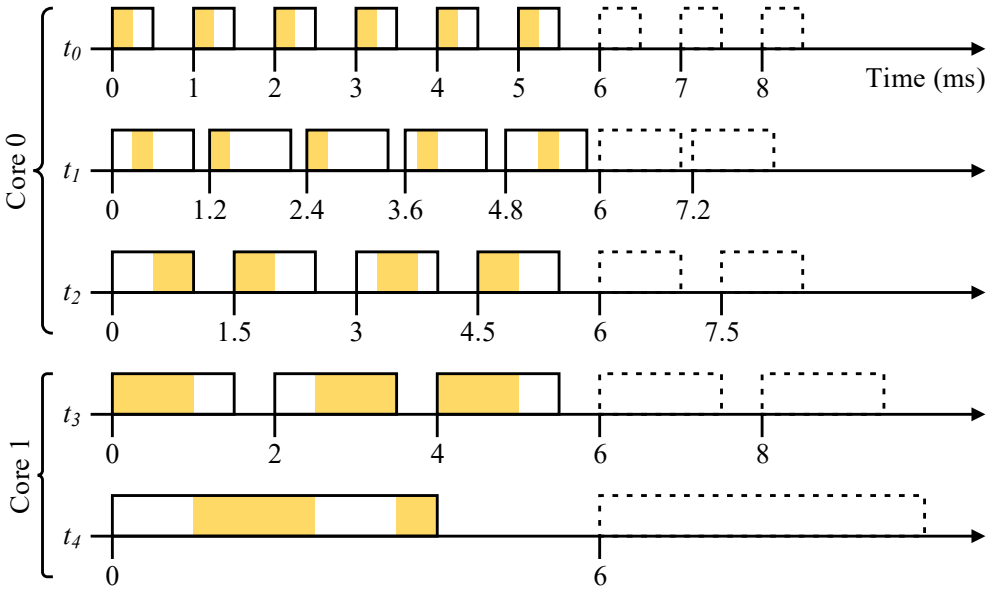


Figure 4: Hyper-period schedule of 6 ms for the tasks in Table 1. Execution times allocated in each LET are indicated by shaded segments.

their last period together. Thus, the duration of the resulting hyper-period schedule is equal to the least common multiple (LCM) of the task periods. At run-time, if the boundaries of multiple LETs occur together, then the writes always precede the reads. This ensures that the latest value of each signal can be read.

System schedulability is demonstrated by constructing a hyper-period schedule that provides enough time for tasks to execute at their WCET during their LET. The guarantee of signal stability and the absence of partial reads by the LET semantics allow tasks to be scheduled preemptively for improved schedulability [KS12]. A LET in the hyper-period schedule contains slack if it is not completely allocated to execute tasks. For Core 1 in Figure 4, task t_3's third LET contains slack. Note that there is no slack in t_3's first two LETs and in t_4's first LET because those time periods are allocated completely to execute t_3 and t_4. If every LET contains slack, then the system's end-to-end response times can be reduced by scaling down the timing parameters of all tasks until a task no longer has any slack. This results in a shorter hyper-period. However, absolute timing behaviour is not preserved by this approach.

2.6 Preservation of LET Communication Semantics

One key benefit of using the LET task model is that its formal semantics [HHK01] facilitates the formal verification [CW96] of a system's functionality and timing behaviour against its

requirements. An implementation that preserves the LET semantics does not need to be verified, since its behaviour would be identical to that of the original design. When given the same sequence of (timestamped) inputs, a semantics preserving implementation and its original design would produce the same sequence of (timestamped) outputs, i.e., the data-flow and its timing are preserved. However, the idealised instantaneous writing and reading of signals at LET boundaries cannot be realised by any implementation; time is always needed. Thus, a correct implementation must ensure that sufficient time is provided to access signals so as to preserve the original data-flow and its timing.

2.7 Use of LET as a Design Contract

The automotive industry is actively exploring [EKQS18] the use of the LET task model as a design contract between control engineers, who demand information on the delays that their control loop could experience, and software engineers, who are responsible for implementing the control algorithms such that they run at their designed rate. The control and software engineers would negotiate on how the control algorithm is to be mapped as sequences of runnables to LET tasks, and on the LET timing characteristics. The mapping has to consider the resource needs of each runnable, which may be restricted to specific processor cores, e.g., signal processing execution units, or peripherals for sensing and actuating. Once the contract is settled, the control and software engineers could start working independently of each other. The control engineers would design their algorithm, knowing the expected end-to-end response times of the final implementation with high confidence. The software engineers could explore different implementation options with minimal risk in affecting the control quality. Consequently, it is undesirable to later modify the runnable-to-task mappings, because the end-to-end response times may be greatly affected, warranting a full redesign of the control algorithm.

Table 2: Categorisation of the semantics preserving protocols reviewed in Section 3

Centralised:
Dynamic buffering protocol (DBP) [STC06]
Temporal concurrency control protocol (TCCP) [WNSV07]
Timed implicit communication protocol (TICP) [KQBS15]
Decentralised:
AUTOSAR implicit communication [AUT17a]
LET point-to-point (PTP) buffering [RNL17, HvHM+16, RNH+15]

3 Related Work on Semantics Preserving Buffering

This section reviews the wait-free buffering protocols that have been proposed for AUTOSAR task communication [AUT17a], and for time-triggered communication based on LET semantics [KS12] and the closely related synchronous-reactive semantics [BCE+03]. A buffering protocol defines the necessary actions that the run-time and tasks need to take to manage and access a buffer's content. The protocol guarantees that the signal writer and readers always access the same buffer elements at disjoint times, and that the freshest value is always read. Typically, a buffer is created for each signal and its value is written by the output of a dedicated task, called the *writer* of the signal. A task that reads the signal's value as input is called a *reader* of the signal. Note that a task can write to or read from multiple signals.

Table 2 categorises buffering protocols as being *centralised* [KQBS15, WNSV07, STC06] or *decentralised* [RNL17, HvHM+16, AUT17a, RNH+15] depending on the buffer's location in memory. Centralised protocols use a buffer that is located in global memory. With decentralised protocols, a signal's value is written to the writer's local buffer, and the readers are responsible for copying the value into their own local buffers. Although centralised protocols can be more memory efficient than decentralised protocols, accessing global buffers can be more time consuming for frequent signal accesses.

3.1 AUTOSAR Implicit Communication

AUTOSAR supports the decentralised buffering of signals via so called *implicit communication* [AUT17b]. For each runnable, the AUTOSAR run-time environment copies its input signals into local variables before the runnable is executed, and writes its output signals after the runnable has terminated. Runnables access their own copy of inputs during execution. Thus, signal stability and the absence of partial reads is guaranteed by the run-time. However, even on the same platform, the run-time does not guarantee the timing or ordering in which the inputs and outputs are copied. Hence, implicit communication is inherently non-deterministic and, thus, unsuitable for preserving LET semantics.

3.2 LET Point-to-Point (PTP) Buffering

Buffering protocols proposed for LET systems are based on a decentralised *point-to-point* (PTP) approach [RNL17, HvHM+16, RNH+15]. These protocols are designed for systems that use priority-based task scheduling, such as OSEK OS [OSE05]. A task's output signal is computed and stored in a local buffer, and only made available at the end of its LET. When a reader of the signal starts its LET, it stores a copy of the signal in its own local buffer. Thus, the collective buffer size for a signal is equal to $R+1$, where R is the number

Table 3: Example timing information (in ms) from Table 1 for the tasks in Figure 1d

Task	Period	WCET
t_0	1	0.25
t_1	1.2	0.25
t_2	1.5	0.5
t_3	2	1
t_4	6	2

of readers and "1" is needed for the writer, although this can be reduced by performing buffer analysis [RNL17, RNH+15] to identify the tasks that do not require buffering for semantics preservation. The analysis also identifies tasks that can share a global buffer without affecting the communication behaviour, resulting in a more centralised protocol.

3.3 Dynamic Buffering Protocol (DBP)

In contrast to LET tasks, where outputs are expected at predefined times, the outputs of synchronous-reactive tasks [BCE+03] are assumed to be produced instantaneously (in zero time) when inputs arrive. However, in any real implementation, tasks need time to compute their outputs. In addition, a task's computation time can vary from one instance to another. Thus, buffering is needed to ensure that tasks read from the correct output instances [NWV08, STC06] in order to preserve the synchronous communication semantics. Sofronis et al. [STC06] propose a *dynamic buffering protocol* (DBP) that is memory optimal in the sense that only the output instances needed for semantics preservation are buffered, with no assumptions made on task activation or completion times. The writing task uses a `next` pointer to track the buffer element that will hold the new value being computed, and a `prev` pointer to track the buffer element of its previously computed output. Each time the writing task is activated, it assigns `next` to `prev`, and an algorithm is executed to find a free buffer element that is not used by a reading task or pointed to by `prev`. The `next` pointer is updated to point to the free buffer element. When a reading task is activated (at the start of its period), it copies the address held in `next`. This address specifies the buffer element that the reading task uses throughout its computation. The address held in `prev` is copied instead if the reading task has a higher priority than the writing task. Buffer elements are freed and reused when their values are no longer needed by the readers.

Figure 5 demonstrates DBP for signal s_0 from Figure 1b, using the task periods and WCETs from Table 3 (i.e., treating them as ordinary tasks without LET semantics). The task priorities, in descending order, are $t_0 > t_1 > t_2 > t_3 > t_4$. Since DBP is designed for single-core platforms, the execution trace assumes rate-monotonic, preemptive scheduling [LL73] on a single core. At 0 ms, after all tasks have been activated, the readers will read from buffer element e_1, even though its value is currently undefined. By the time the readers are scheduled for execution, t_0 has written the value 1 into buffer element e_1. We see that buffer element e_1 could not be reused during t_4's entire period. At $2ms$, the buffer is fully utilised because element e_0 holds the previous value, element e_1 is being read by task t_4, and element e_2 is needed for the writer's next value that task t_3 reads. Even after t_2 is preempted at 4.8 ms, it continues to correctly read value 5, instead of the next value 6 computed by t_0.

DBP can be configured to store up to k previous values of a signal, which is useful when tasks need a sliding window of values for signal processing [BDM02], or need to access pre-

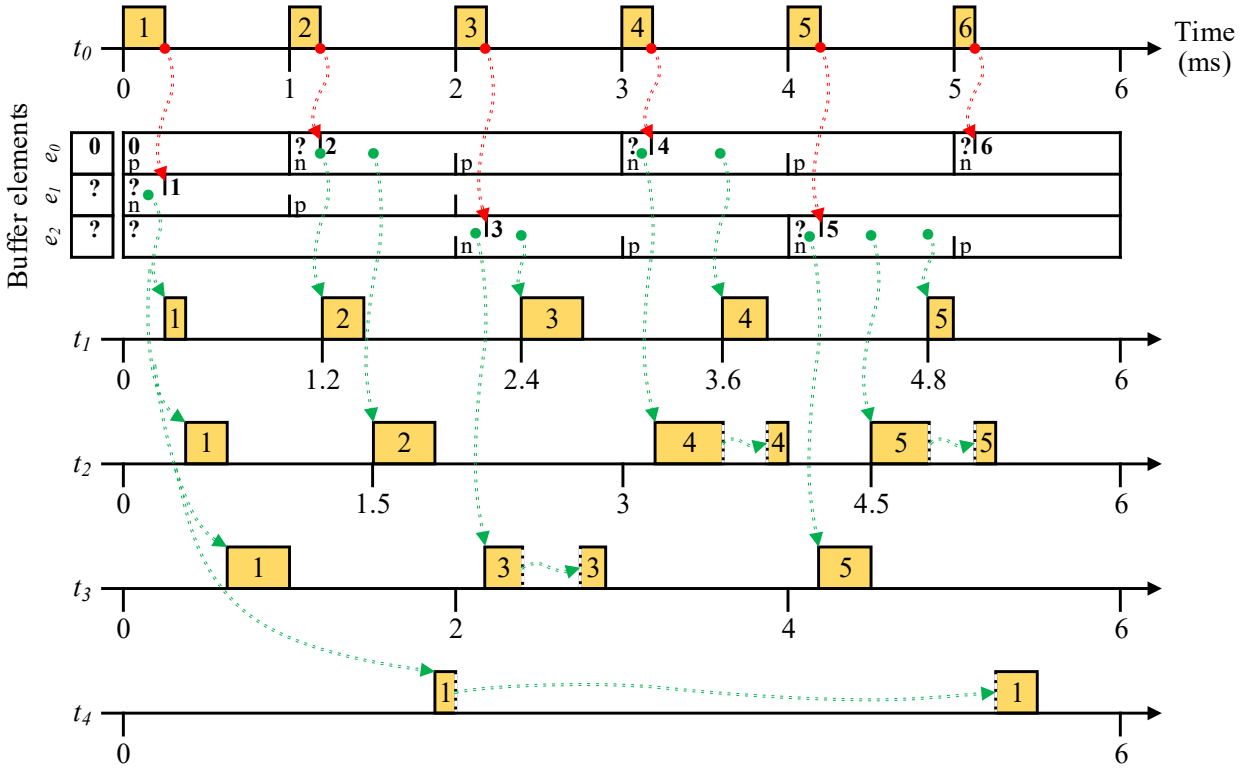


Figure 5: Example execution of the tasks in Table 3 using DBP for signal s_0 from Figure 1b. For the first 6 ms, the contents of signal s_0's buffer are displayed below the writer t_0. Each buffer element is shown as a row, containing its value ("?" if the value is being computed) and whether it is being referenced by the writer's `next` (n) or `prev` (p) pointer. Changes to a buffer element's value or to the writer's pointer references are demarcated by solid vertical lines. Writes and reads are drawn as dotted arrows going into and out of the buffer, respectively. The values written and read by the tasks are shown inside their respective LETs. Task preemptions are indicated by dotted vertical lines.

vious values to correctly implement software pipelining [MRR12]. Moreover, DBP supports the over- and under-sampling of signals when tasks of different periods communicate. Under dynamic task scheduling, a lower bound for a signal's buffer size is calculated [STC06] as $R_{lp} + k + 1$, where R_{lp} is the number of lower priority readers, and k is the number of previous values to retain. For Figure 5, a buffer size of 4 would be calculated, although only a size of 3 is actually needed.

3.4 Temporal Concurrency Control Protocol (TCCP)

Wang et al. [WNSV10, WNSV07] provide several OSEK-compliant implementations for DBP and analyse their costs in terms of required memory and execution time, given in Table 4. The main decider for the required memory and execution time is the algorithm for finding a free buffer element. For the constant-time implementation, an auxiliary linked list is used to track the free buffer elements, leading to a higher memory requirement than the linear-time implementation, which simply iterates through the entire buffer until a free element is found. Wang et al. [WNSV10, WNSV07] also describe a *temporal concurrency control protocol* (TCCP) that uses a circular buffer [KR93] to store a signal's values in consecutive (chronological) order. Thus, finding a free buffer element only involves incrementing `next` to point to the next buffer element. For TCCP, the buffer size is bounded by the number of writes that could occur during the longest task period among the readers. If TCCP had been used for Figure 5, then a buffer size of 7 would be calculated.

3.5 Timed Implicit Communication Protocol (TICP)

The *timed implicit communication protocol* (TICP) [KQBS15] extends AUTOSAR implicit communication by tagging each written value with a monotonically increasing timestamp. To preserve the communication semantics, each reader is responsible for finding the value with the correct timestamp. In any real implementation, the memory for storing each timestamp is bounded, posing a limit on the system's run-time before a timestamp overflow occurs [ST00]. No algorithms are suggested to find a free buffer element for the writer, to find the correct timestamped values for the readers, or to handle bounded timestamps. TICP appears to be similar to DBP, except that DBP implicitly maintains the necessary timestamp information with the `prev` and `next` pointers.

3.6 Related Buffering Protocols

Other buffering protocols have been proposed, but are not directly applicable to LET communication. First in, first out (FIFO) buffering [Hab72] is used in point-to-point signal communication, where a reader needs to receive all values computed by a writer. The reader consumes (reads and then clears) the oldest value in the buffer. However, FIFO buffering is unsuitable when tasks have different periods, because it can lead to buffer over- or under-flow. Similar to FIFO buffering is lossless [YKRB14] and synchronous data flow (SDF) buffering [LM87]. In lossless buffering, the reader consumes all values in the buffer each time it is activated. In SDF buffering, each time a task is activated, it consumes or writes a fixed number of values into the buffer. We do not consider SDF or lossless buffering in this work because current automotive systems do not require such communication behaviour.

Table 4: Memory and execution time costs for DBP, TCCP, and PTP, where B is the number of buffer elements, R is the number of readers, R_{lp} is the number of lower priority readers, k is the number of previous values to retain, p_R^{max} is the maximum task period among the readers, and p_W^{min} is the minimum task period of the writer

	Memory		Time to find buffer element	
Buffering protocol	**Buffer**	**Auxiliary**	**for writing**	**for reading**
Linear-time DBP [WNSV07]	$B = R_{lp} + k + 1$	$3R + B + 2$	$O(B)$	$O(1)$
Constant-time DBP [WNSV07]		$3R + B + 3$	$O(1)$	
Constant-time TCCP [WNSV07]	$\left\lceil \frac{p_R^{max} + p_W^{min}}{p_W^{min}} \right\rceil + k$	$2R + 2$		
Constant-time PTP [RNH+15]	$R + 1$	0		

3.7 Discussion

The memory and time trade-off highlighted by Table 4 is that a faster buffering protocol needs to store more information about the tasks at run-time, while a slower protocol needs time to reconstruct the information every time it is invoked. The semantics preserving DBP, TCCP, and TICP protocols have been designed with priority-based, preemptive task scheduling in mind, and make no assumptions about task activation and completion times. Moreover, they assume that a signal has only one writer, whereas real automotive software can have signals with multiple writers. Only task periods are required, which are used to derive task priorities. Therefore, buffer management algorithms need to be executed at run-time, e.g., to find a free buffer element for the writer, and to find the correct signal snapshot to read. These buffering protocols assume a more general task model than LET, and can be adapted to preserve LET semantics. However, by design, DBP and TCCP are limited to single-core platforms, because tasks are assumed to execute sequentially and never in parallel. By observing that LET tasks have precisely defined input reading and output writing times, their computation and buffer accesses can be statically scheduled so as to avoid the need to manage the buffers at run-time. Moreover, exact buffer sizes can be computed for each signal by inspecting the static schedule (see Section 7.1).

It should be noted that the actual buffer size needed by DBP is never greater than that of TCCP [STC06]. However, depending on the task periods, the calculation of a lower bound on the buffer size needed by DBP can sometimes be worse than that of TCCP, leading to the over-provisioning of buffer memory. Natale et al. [NWV08] reduce the calculated lower bounds for DBP by observing that readers, with slightly longer periods than the writer's period, access the same subset of buffer elements. Thus, the reading tasks are partitioned into *faster tasks* and *slower tasks*, and a lower bound is calculated for each set. The lower bounds are summed together to obtain a final lower bound. For Figure 5, an improved buffer size of 3 would be calculated, equal to what is actually needed.

4 Related Work on Optimising AUTOSAR Designs

This section reviews the approaches that have been developed to optimise the end-to-end response times, memory consumption, and processor utilisation of AUTOSAR designs. We begin by reviewing the approaches developed for single-core AUTOSAR designs [FNG+09, ZN12, ZNZ14] that are based on traditional operating system tasks. When moving to multi-core designs [FLSN14, WMM+13, SCCM15, HZN+14], the approaches need to consider the resource contentions that arise from parallel execution, e.g., on the system bus, shared memories, and peripherals. We end by reviewing the optimisation approaches for LET-based AUTOSAR designs [RNH+15, HvHM+16, FFPT05, BKU16, RNL17, BPBN17], which is a relatively new research area. The approaches developed for multi-core AUTOSAR designs cannot be applied directly because greater care is needed to preserve the LET semantics. Moreover, LET tasks are typically compiled [HK07] for execution on a virtual *embedded machine*, which hinders the application of performance-related optimisations on a real platform. Nevertheless, several implementation strategies for real platforms have been proposed [RNH+15, HvHM+16, KSU16, RNL17].

4.1 Optimising Traditional AUTOSAR Designs

The optimisations developed for single-core designs [FNG+09, ZN12, ZNZ14] typically assume the fixed-priority scheduling of periodic operating system tasks. Each periodic task contains one or more runnables, whose WCETs are known at design time. Tasks communicate via signals, and each signal is assumed to have a dedicated writing task. Under these assumptions, Zeng and Natale [ZN12] minimise the total memory needed to maintain context-switches on the stack and to manage signal communication. Heuristics and *mixed-integer linear programming* (MILP) [BGG+71] are used to optimise, for each signal, the use of locks or wait-free protocols (DBP or TCCP) as a means of trading off memory consumption with execution time [FNG+09]. If a lock is used, then the signal's access time is modelled as a critical section to account for the potential blocking time. If a wait-free protocol is used instead, then the signal's access time is modelled as additional instructions in the runnables to reflect the protocol's overhead. Stack usage is minimised through the concept of *preemption thresholds* [WS99], where a task is scheduled based on its normal priority, but is executed at a higher priority that is equal to that of its executing runnable. This elevation in task priority helps one to minimise the occurrence of task preemptions. An optimal assignment of runnable priorities and execution orders are found for each task, such that tasks with shorter deadlines can still preempt tasks with longer deadlines, thereby ensuring task schedulability. In a later work [ZNZ14], the merging of tasks is considered to improve the assignment of runnable priorities and execution orders.

The optimisations developed for multi-core or multi-processor designs make the same assumptions as those for single-core designs. Additionally, the use of partitioned scheduling is assumed, where tasks are statically assigned to a core and cannot migrate to another core at run-time. Several resources (e.g., the system bus, memory modules, or peripherals) may also be shared among the cores. Faragardi et al. [FLSN14] propose a heuristic that uses simulated annealing [KSU83] to find good task-to-core allocations such that the overall runnable communication time is minimised. Runnables involved in the same event-chain are assigned to the same task, and their ordering in the event-chain becomes their execution order. A runnable is duplicated if it is involved in multiple event-chains. Different communication delays are modelled for runnables that belong to the same task (shortest delay), to the same core, or to different cores (longest delay). Tasks are merged if their runnables communicate with each other, because this further reduces the communication delays. The

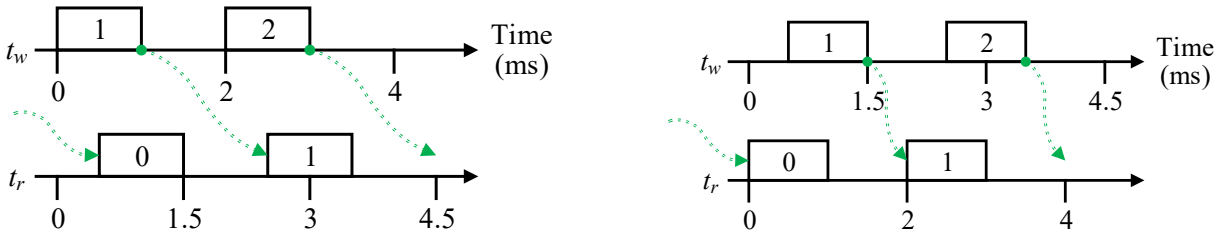
proposed heuristic ensures that feasible solutions satisfy the end-to-end response times of the design's event-chains. However, no attempt is made to minimise memory consumption and data protection is not considered.

Wozniak et al. [WMM+13] consider a multi-processor architecture where each processor can only access local memory, and must communicate with others over a shared bus. The optimisation goal is to minimise end-to-end response times and memory consumption. For tasks that reside on the same processor, time-consuming (locks) and memory-consuming (wait-free) data protection mechanisms are selected for each signal. When locks are used, the potential blocking time across the tasks is considered. Bus throughput, where only a limited amount of data can be transferred on the bus at any given time, is maximised by allocating runnables to the same task (core) if they communicate together. The end-to-end response times are calculated by summing the worst-case response times of the runnables and the worst-case access times of the signals. Unlike Faragardi et al. [FLSN14], runnables involved in multiple event-chains do not need to be duplicated, and runnables in the same event-chain can be allocated to different tasks and cores. Thus, changing the runnable execution orders can have a significant impact on the end-to-end response times. To support the modelling of heterogeneous architectures, each runnable has a vector of WCETs, where each WCET corresponds to a specific processor. In addition, each signal has a vector of worst-case access times, where each access time is for a specific combination of buses. Although the modelling of heterogeneous architectures increases the applicability of the proposed optimisation approach to realistic systems, it also causes the design space to explode. Thus, in addition to solving the optimisation problem exactly with MILP, Wozniak et al. [WMM+13] provide a heuristic based on a genetic algorithm [Gol89] that gives approximate solutions, i.e., possibly suboptimal, in less time.

Saidi et al. [SCCM15] investigate the problem of minimising task communication times while statically load balancing a homogeneous multi-core processor. *Integer linear programming* (ILP) [PS82] is employed to find optimal runnable-to-core allocations such that runnables that communicate frequently are on the same core, and that the absolute load difference between the cores is minimal. Han et al. [HZN+14] study the implementation of lock-based and wait-free protocols and present detailed measurements of their memory and time overheads. Based on their experimental data, they propose a greedy heuristic that selects a data protection mechanism for each signal in a system with the aim of reducing memory consumption. The heuristic begins by assuming that all signals use a wait-free protocol (DBP or TCCP). The signals are then ordered based on how much memory would be saved if a lock is used instead. The signal with the largest memory saving is picked, and its protection mechanism is switched to a lock if the tasks remain schedulable. The remaining signals are analysed in the same manner, in decreasing order of memory saved. This heuristic has linear-time complexity, but a locally optimal selection for a signal could force suboptimal mechanisms to be selected for subsequent signals.

4.2 Optimising LET Designs

There is a strong desire by the automotive industry to reuse existing legacy AUTOSAR software alongside new software on multi-core platforms. The timing behaviour of the legacy software must be preserved when modernised for multi-core platforms. Resmerita et al. [RNH+15] showed that legacy software components can be wrapped inside LET tasks and be parameterised according to the observed timing behaviour of the original components. The timing of the modernised software behaved nearly identically to that of the original software. A main concern with this approach is the need to introduce PTP buffers and



(a) Both t_w and t_r need their own buffer.

(b) Only t_w needs a buffer.

Figure 6: Pruning of PTP buffers for a signal between writer t_w and reader t_r. The values written and read by the tasks are shown inside their respective LETs.

so-called *drivers* that perform the buffered reads and writes needed to respect the LET semantics. To keep memory requirements to a minimum, buffer analysis is performed to prune away buffers that are not actually needed to preserve the LET semantics. Consider the scenario in Figure 6a, where writer t_w and reader t_r communicate via a signal. Assume that the inputs and outputs of the tasks are not buffered. If t_w can complete its execution and write the signal's next value before t_r has begun to execute, then t_r will read the wrong value. Hence, t_w's output needs to be buffered. Since t_w's LET ends during t_r's LET, the signal's value could become unstable during t_r's execution. Thus, t_r's input also needs to be buffered. Figure 6b shows another scenario where t_r's LET is ahead of t_w's. In this case, only t_w's output needs to be buffered to prevent t_r from reading the signal's next value.

Resmerita et al. [RNL17] extend their buffer analysis to multicore platforms, where task-to-core allocations are assumed to be given. Their original buffer analysis is performed in the design phase, and additional analysis is performed in the deployment phase where additional information, such as task-to-core allocations and task priorities, are known. In the deployment phase, buffers can be removed for tasks that are scheduled to execute sequentially on the same core. For example, if t_w and t_r in Figure 6b are allocated to the same core and t_r has a higher priority than t_w, then t_r always finishes executing before t_w is scheduled to execute. Hence, communication from t_w to t_r is purely sequential and no buffers are needed. Signal buffers on the same core can be reused if their lifetimes do not overlap. However, buffer analysis is not applied across the cores because a total order cannot be derived among all tasks.

Farcas et al. [FFPT05] focus on scheduling the transfer of signal values between multiprocessors. The main observation is that a task's output can be suppressed if it will be overwritten by a fresher output before any reading task can start its LET. Moreover, a signal's value does not need to be transferred instantaneously, so long as it is available by the time a reading task starts its LET. This enables bursts of bus activity, due to the alignment of multiple LET boundaries, to be smoothed out over time.

Biondi et al. [BPBN17] analyse a realistic AUTOSAR design from the Formal Methods for Timing Verification (FMTV) challenge [HDK+17], and proposed an MILP formulation and corresponding genetic algorithm that minimises the response times of runnables through the allocation of signals to local and global memory modules. When LET semantics is assumed for task communication, the end-to-end response times of the event-chains in the design are invariant to the signal-to-memory allocations. Nevertheless, processor utilisation would be reduced as a side-effect of minimising the runnable response times. Under AUTOSAR *explicit communication*, where runnables directly access the signals without buffering, the end-to-end response times are shorter with the optimised signal-to-memory allocations than with the original allocations provided by FMTV challenge.

A LET task's response time can be improved by shortening its LET duration, such that outputs become available earlier. However, this decreases the task's schedulability because its computations must be scheduled within a shorter LET duration. Using this observation, Bradatsch et al. [BKU16] perform response time analysis to determine tasks that can have shorter LET durations, while ensuring that the system remains schedulable. Although this can reduce the end-to-end response times of some event-chain instances, the worst-case remains unchanged and the data-flow is not preserved.

4.3 Discussion

The optimisations developed for traditional AUTOSAR designs (based on operating system tasks) are not concerned with preserving the data-flow or timing determinism; the design itself never had such properties. Consequently, more design parameters can be altered when compared to LET-based optimisations, e.g., runnable execution order, runnable grouping, and using non-semantics preserving buffering mechanisms. Correctly optimised LET implementations must preserve the data-flow and timing of their designs. An important aspect not addressed by the related work is that automotive systems are memory constrained. Thus, a feasible solution must also guarantee that signals (and their buffers) can fit within their allocated memory module. Only Wozniak et al. [WMM+13] consider the related issue of maximising bus throughput.

In this work, we are not concerned with finding alternative LET designs, but with finding better implementations of the same design. Thus far, LET implementations have only employed PTP buffering (see Section 3.2) and fixed-priority task scheduling. There does not appear to be any optimisation approaches that cater to statically scheduled LET tasks, e.g., base-period or hyper-period scheduling (see Section 2.5), or to the selection of several buffering mechanisms for each signal. We offer more extensive optimisations for LET-based AUTOSAR software: apart from pruning unnecessary buffers at the design and deployment phases, suppressing unnecessary signal writes, and merging buffers, we can also (1) select for each signal the use of PTP or a static version of DBP, (2) cater to signals with multiple writers, (3) leverage data-age constraints to suppress unnecessary writes, (4) allocate LET tasks to heterogeneous cores, (5) allocate signal buffers to memory modules with limited capacities, and (6) suggest scheduling hints to a hyper-period task scheduler. Our approach is supported by a set of heuristics, algorithms, and an MILP formulation for which we provide a genetic algorithm when faster solving times and approximate solutions are preferred.

Resource allocation and scheduling problems are known to be NP-hard, so heuristics, such as genetic algorithms or simulated annealing, are needed to find (sub)optimal solutions for large systems within a reasonable amount of time. Such heuristics attempt to explore the design space in an intelligent manner by repeatedly creating new candidate solutions from initial or prior solutions, and ranking the candidates based on an objective function (called *fitness* in genetic algorithms and *cost* in simulated annealing). During the exploration, some candidate solutions may be unable to satisfy all constraints (e.g., memory and processor utilisations), but may be close with only some design variables needing minor adjustments. Thus, it is important to incorporate a notion of *penalty* into the objective function to penalise infeasible solutions by how badly they have exceeded the constraints.

5 System Model

This section describes our hardware and software model of AUTOSAR systems to place our proposed LET buffering optimisations into perspective. It includes our assumptions on LET task executions, task communications, task scheduling, signal buffers, processor cores, memories, and buses. We assume that all memory sizes and timing information are expressed in consistent units, e.g., as bits and in milliseconds, respectively.

5.1 Software Model

The software model is concerned with LET tasks and the hyper-period scheduling approach, and the signals and their usages by tasks, and the signal buffers.

Tasks. An AUTOSAR design consists of runnables and we assume that their allocation to LET tasks is given. If a runnable is shared among multiple software components, then it is duplicated and given a unique name. Instances of the LET tasks are statically scheduled under the hyper-period approach, and we optimise AUTOSAR designs at level of task instances. Thus, an AUTOSAR design contains a set of LET tasks, $t_a = \langle period, letStart, letEnd, acc \rangle \in T$, where each task has a unique name "a", a period, LET start and end times that are relative to the start of the task's period, and the signals the task accesses. An instance of task t_a, i.e., $t_a^i = \langle periodStart, letStart, letEnd \rangle$, has an instance number $i \in \mathbb{N}_0$ that starts from 0, an absolute start time for its period, and absolute LET start and end times:

$$t_a^i.periodStart = i \times t_a.period \tag{1}$$

$$t_a^i.letStart = t_a^i.periodStart + t_a.letStart \tag{2}$$

$$t_a^i.letEnd = t_a^i.periodStart + t_a.letEnd \tag{3}$$

Only LET tasks with non-overlapping instances, i.e., LET start and end times that are within their period, are considered. Overlapping instances could be modelled by separate LET tasks. If multiple task instances start or end their LET at the same time, then the end of all LETs always complete before the start of any LET can proceed. This ensures that tasks always read the freshest value of signals. For each signal accessed by a task, an upper bound n can determined by examining the runnable's code, i.e., $acc = \{\langle s, n \rangle\}$.

Task schedules. Without loss of generality, we assume that all tasks have zero initial offsets, i.e., all tasks start their initial period together, in order to simplify the construction of hyper-period schedules. The duration of a hyper-period schedule is the least common multiple (LCM) of all task periods, i.e., $hp = \text{LCM}(\{t_a.period \mid t_a \in T\})$. We call the hyper-period schedule shown in Figure 4 a *physical task schedule*, because it contains deployment information such as task-to-core allocation and execution times. Preemptions are allowed in the schedule, but task migrations are disallowed. A *logical task schedule* is one that only contains information about the logical timing of the tasks, i.e., only their LET start and end times.

Signals. Tasks communicate via signals, $s = \langle size, n, style \rangle \in S$, where each signal has a data size, an SBP buffer size of n elements determined during buffer analysis (see Section 7.3), and is associated with a *local* or *global* programming style. A local programming style, see, e.g., Figure 7a, means that the signal's intermediate value is stored in a writer's local variable before its final value is written to the signal. With a global programming

```
void task(int input) {
  int nextOutput = 0;
  if (input == 0)
    nextOutput = comp();
  output = nextOutput;
}
```

(a) Local programming style.

```
void task(int input) {
  output = 0;
  if (input == 0)
    output = comp();

}
```

(b) Global programming style.

Figure 7: Programming styles of signals.

style, e.g., Figure 7b, the intermediate values are stored in the signal itself. Hence, writes to a signal with the global programming style cannot and must not be suppressed. The functions $\mathrm{W}(s)$ and $\mathrm{R}(s)$ return the set of tasks that write to and read from signal s, respectively. Signal dependencies between the tasks and data age constraints are assumed to be given as input by the designer. A data age constraint, $t_a \xrightarrow{s,\ \delta} t_b \in \mathit{DataAges}$, specifies that the value of signal s read by t_b must not have been written by t_a more than δ time units ago. The default behaviour of reading the freshest possible value is modelled by $\delta = t_a.\mathit{period}$.

Multiple signal writers. In automotive software, it is possible for a signal to have multiple writers. This could arise when a task is split across several cores for better load balancing. It can also occur in legacy software, where tasks communicate over shared memory by explicitly writing to the same signal. Thus, unlike related work, we consider signals that have multiple writers. If the writers have LETs that end at different times, then the signal always has a unique value. However, if two writers have LETs that end at the same time, then the signal's value is indeterminate. To ensure determinism, the designer specifies which writer instances will define the signal's value at any given time. Let $\mathit{allWI}_a \in \mathit{AllWI}$ where $\mathit{allWI}_a = \{t_a^i \mid 0 \leq i < \frac{hp}{t_a.\mathit{period}}, i \in \mathbb{N}, t_a \in \mathrm{W}(s)\}$ be the set of all possible writer instances of signal s during one hyper-period. The designer selects a subset of writer instances, $\mathit{selectedWI}_a \in \mathit{SelectedWI}$ where $\mathit{selectedWI}_a \subseteq \mathit{allWI}_a$, such that all instances have unique LET end times, i.e., $\forall t_a^i, t_b^j \in \mathit{selectedWI}_a : t_a^i \not\models t_b^j$ implies $t_a^i.\mathit{letEnd} \neq t_b^j.\mathit{letEnd}$.

Signal usage. To facilitate buffer analysis, we record the time intervals that each signal is used by the tasks. We say that a task begins to use a signal when the task starts its LET, and finishes using the signal when the task ends its LET. The start and end of each usage is labelled $\mathcal{R}^{start}$ and $\mathcal{R}^{end}$ when the task is a reader, and $\mathcal{W}^{start}$ and $\mathcal{W}^{end}$ when the task is a writer. The uses of signal s, uses_s, is a set of tuples $\langle t_a^i, \sigma, \phi \rangle$, each defining a task instance's usage $\phi \in \{\mathcal{R}^{start}, \mathcal{R}^{end}, \mathcal{W}^{start}, \mathcal{W}^{end}\}$ at an absolute timestamp σ in the hyper-period. Section 7.2 defines how uses_s is actually built. The function $\mathrm{GetUses}(\mathit{uses}_s, \sigma_0, \sigma_1, \phi)$ returns from uses_s the usages with label ϕ and timestamp σ such that $\sigma_0 < \sigma < \sigma_1$. The function $\mathrm{GetEarliestUses}(\mathit{uses}_s)$ returns from uses_s the usage(s) with the earliest timestamp.

Buffers. We consider two buffering protocols: PTP (see Section 3.2) and SBP (see Section 7.1). Each signal is associated with a buffer, $\mathit{buff}_s = \{e\} \in \mathit{Buffs}$, which itself is a set of abstract memory elements e. For buffer analysis, each buffer element $e = \{t_a^i\}$ records the set of task instances that can be using it. A signal's buffers are managed at run-time by a statically computed *buffering schedule*, $\mathit{buffSch}_s \in \mathit{BuffSchs}$, that allocates a task instance to a buffer element, i.e., $\mathit{buffSch}_s = \{\langle t_a^i, e \rangle\}$. There may be intervals

in the hyper-period when a signal's buffer content is being used by tasks. These intervals have absolute start and end times, i.e., $interval = \langle start, end \rangle$, within the hyper-period, and together they define the buffer's *lifetime*, i.e., $lifetime_s = \{interval\}$, and $lifetime_s \in Lifetimes$. Pairs of signals, e.g., s_0 and s_1, with non-overlapping buffer lifetimes are stored in $disjBuffs = \langle s_0, s_1 \rangle \in DisjBuffs$ as candidates for buffer merging.

5.2 Hardware Model

The hardware model is concerned with the organisation of the processor and its cores, the memory modules and the communication buses, and details relating to task execution times and memory access times.

Cores and memory modules. Our model of a heterogeneous multi-core processor with non-uniform memory accesses (NUMA) is based on the AURIX TC27xC series of multi-core processors from Infineon Technologies AG [Inf14], depicted abstractly in Figure 1d. Each core $c \in C$ has a pathway to one or more memory modules $m = \langle size \rangle \in M$ of fixed capacity ($size$). We use pathways to abstract from the traversal of one or more communication buses. All memory modules are assumed to have the same data width. Each pathway, $path = \langle c, m, l \rangle \in Paths$, between core c and memory module m has a fixed latency l. If multiple pathways exist between c and m, then only the one with the shortest latency is relevant. Such non-uniform access times allow the buffers to be placed *closer* to the writers or to the readers.

WCETs. To focus on the buffer optimisation of AUTOSAR designs, we assume that the instructions executed by a core are fetched entirely from its program memory cache [Inf14]. Thus, buffer access times are isolated from interferences due to instruction fetching. Because worst-case execution timing (WCET) analysis [WEE+08] is not the focus of this work, we assume that the implementor provides the following timing information as input:

- WCET $wcet_{sbp}.c$ to manage an SBP buffer element on core c.
- WCET $wcet_{ptp}.c$ to prepare to copy a signal's value on core c.
- WCET $wcet_{cs}.c$ to perform a context-switch on core c.
- WCET of task t_a on core c, excluding buffering overheads. All instances of a task are assumed to have the same WCET.
- Maximum latency l of each pathway.
- Number w of transfers over a pathway to write or read a signal's value between core c and memory m. This is calculated by dividing the signal's data size by the data width of the memory modules.

We update our definition of tasks and signals to include the additional timing information: $t_a = \langle period, letStart, letEnd, acc, instr \rangle$, where $instr = \{\langle c, wcet \rangle\}$ defines the task's WCET on core c; and $s = \langle size, n, w, style \rangle$, where w is the number of pathway transfers needed.

6 Overview of Proposed Buffering Optimisations

The overall optimisation approach is to minimise the required buffer sizes, the frequency of buffer writes, and the time that has to be allocated for task execution. For each signal, the point-to-point (PTP) buffering protocol (see Section 3.2) or the static buffering protocol (SBP, see Section 7.1) can be selected. The optimisation approach is constrained by the need to allocate tasks and buffers without over-utilising the cores and memory modules, respectively, and to preserve the data-flow and timing of the LET design. A safe but pessimistic approach is taken, where all signal writes and reads are assumed to require buffering, and buffer analysis is performed to safely eliminate unnecessary buffering.

Similar to Resmerita et al.'s approach [RNL17], we apply separate optimisations at the design and deployment phases. The design optimisations are platform independent because they ignore resource allocations and only consider the LET start and end times, i.e., the optimisations work on the logical task schedule. The optimisation results at design phase remain valid in the deployment phase, where a specific hardware platform is chosen and the resource constraints become known. Optimisations are applied to decide on the resource allocations, e.g., for task-to-core, buffer-to-memory, and execution time allocations, i.e., the optimisations work on the physical task schedules. The information that has to be provided as input by the designer and implementor are:

- The software model:
  - logical task schedule, constructed from the timing parameters of the LET tasks,
  - maximum number of context-switches that each task could experience,
  - selected writer instances for resolving race conditions,
  - maximum number of signal accesses by each task,
  - signal dependences between the tasks,
  - data age constraints, and
  - worst-case execution times for each processor core.
- The hardware model:
  - processor cores,
  - memory modules and their capacities, and
  - pathways and their latencies.

Our proposed buffer optimisations can be broken into six major steps, depicted in Figure 8. Design phase optimisations are applied in the first step, and deployment optimisations in the remaining five steps.

Step 1 (see Section 7): The SBP buffering protocol is applied over the logical task schedule to determine the actual buffer sizes and to construct a buffering schedule for each signal. The timing of the LET tasks and their data age constraints are used to identify unnecessary signal writes. The buffering schedules are constructed such that the required buffer memory and buffer management (e.g., the updating of buffer indexes at run-time) are kept to a minimum.

Step 2 (see Sections 8.3 and 8.4): The selection of a buffering protocol for each signal, task-to-core allocations, and buffer-to-memory allocations are decided by an MILP

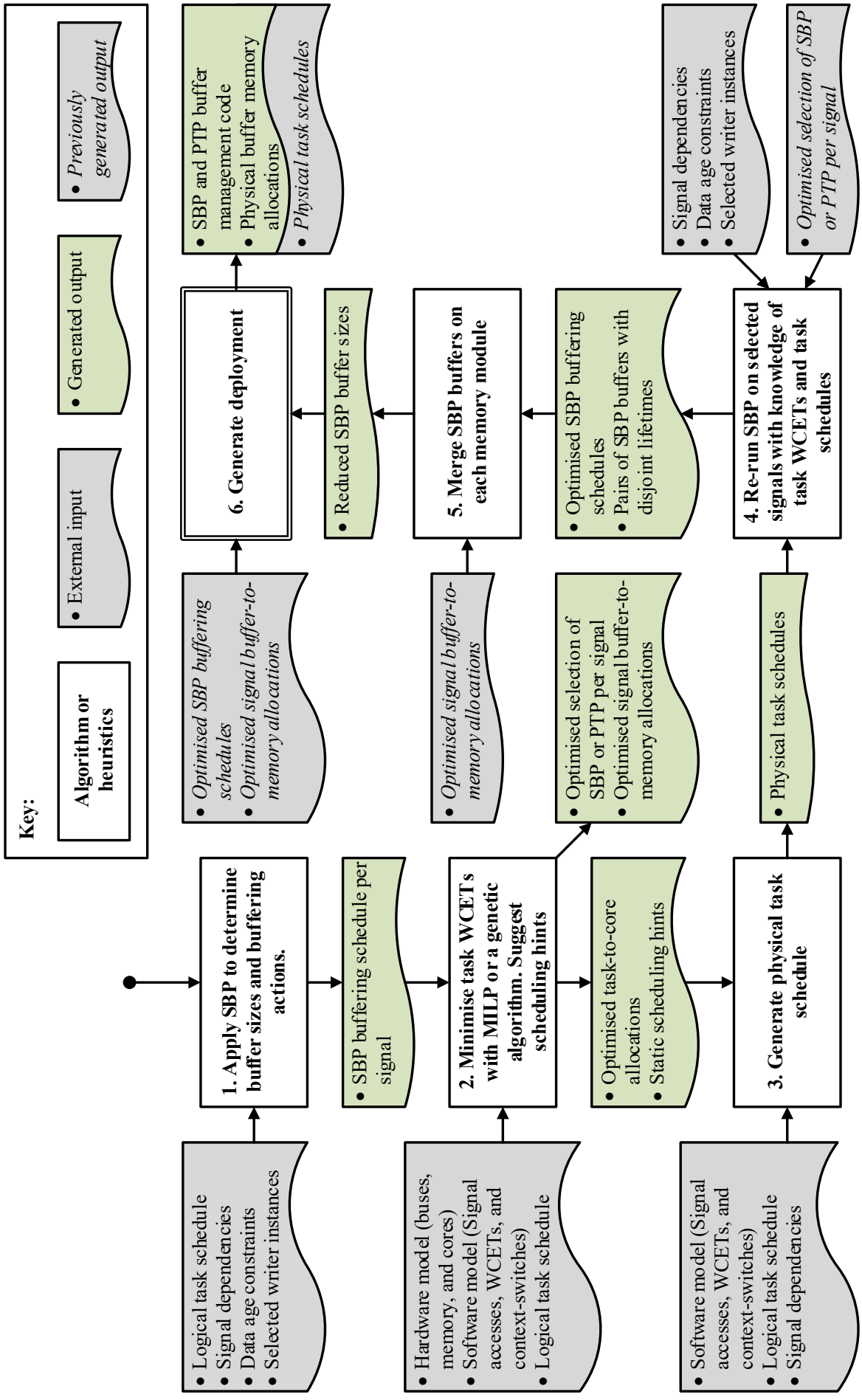


Figure 8: Overview of our approach to optimising LET buffering.

formulation, with the optimisation goal of minimising task execution times. A feasible solution ensures that all tasks are schedulable, and that all memory modules have sufficient space for their allocated signals and buffers. A corresponding genetic algorithm is provided in the case that faster solving times are preferred at the cost of possibly suboptimal solutions. Scheduling hints, e.g., schedule writer t_a as late as possible, are generated for the task scheduler, such that buffer memories can be optimised further in Step 4.

Step 3 (see Section 8.5): The optimised task-to-core allocations and scheduling hints are used to (statically) generate a physical task schedule for each processor core. These physical task schedules are used in the next step to further optimise the SBP buffering schedules. End-to-end response times can be improved by scaling down the entire task schedule until a task has zero slack.

Step 4 (see Section 8.6): The physical task schedules provide concrete timing information on when a task starts executing and the latest time that it finishes executing. Because these start and end times are likely to be shorter than the tasks' LETs, better buffering schedules could be constructed. Hence, Step 1 is repeated with the physical task schedules and new buffering schedules are synthesised for the signals that have been selected to use SBP. In addition, the SBP buffers with disjoint lifetimes are identified.

Step 5 (see Section 8.7): To further reduce the memory needed for the SBP buffers, those with disjoint lifetimes are merged together if they have been allocated to the same memory module.

Step 6 (see Section 8.8): Together with the generated physical task schedules, and the buffer-to-memory allocations, the deployment of the AUTOSAR software is finalised by generating the PTP and SBP buffer management code of each task.

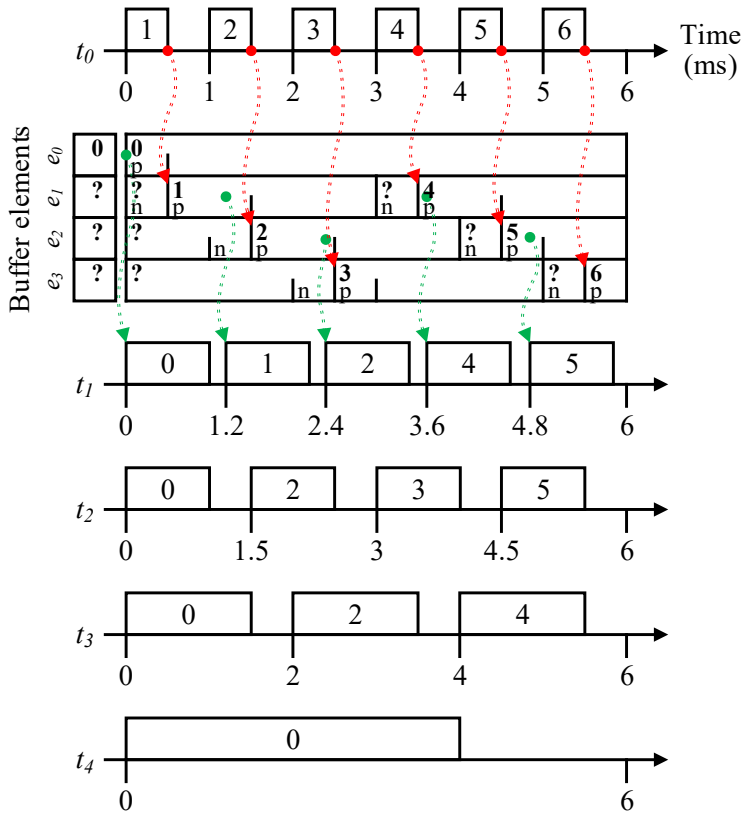


Figure 9: Applying SBP on the logical task schedule of Table 1 for signal s_0 from Figure 1b.

7 Design Phase Optimisations

The design phase optimisations are platform independent and are applied to the logical task schedule. As mentioned in Section 3.7, the DBP [STC06], TCCP [WNSV07], and TICP [KQBS15] buffering protocols can be adapted to preserve the LET semantics. Although DBP and TCCP are limited to single-core platforms under dynamic scheduling, they can be applied to multi-core platforms when static scheduling is used. In this work, we only consider DBP because its buffering capabilities subsumes those of TCCP and TICP.

This section describes the workings of the SBP buffering protocol (see Section 7.1), the identification of buffer writes that can be suppressed without compromising the LET semantics (see Section 7.2), and how the SBP buffering protocol is used to construct a buffering schedule (see Section 7.3).

7.1 LET Static Buffering Protocol (SBP)

The DBP buffering protocol is designed for synchronous-reactive tasks, which read their inputs when they start executing, and write their outputs as soon as they are computed. This is not the case for LET tasks, because signal reading and writing only occurs at LET boundaries. Hence, for LET, a writing task must find a free buffer element when it starts its LET, and only update its `prev` pointer when it ends its LET. Figure 9 illustrates the adapted DBP buffering protocol for signal s_0 from Figure 1b. The use of task priorities in DBP for correct buffering is irrelevant for LET tasks because, in LET, it is only important that all signal writes complete before any reads. Hence, DBP is simplified by assuming that a writing task has higher priority than any reading task.

By scheduling LET tasks with the hyper-period approach, the allocation of and accesses to buffer elements can be statically analysed and scheduled. This is achieved by running

the adapted DBP over the logical task schedule and recording the buffer elements that are allocated to each task instance as a schedule. For example, the trace of the buffer elements in Figure 9 is in essence a buffering schedule. Note that the last value written into a buffer becomes the signal's initial value for the next hyper-period. In Figure 9, we see that the last value is written into element e_3, rather than the initial element e_0. Thus, at the end of each hyper-period, a short routine is needed to copy the last written value into the initial element.

We call the adapted DBP the *static buffering protocol* (SBP), and it is presented as pseudocode in Algorithms 1 and 2. The algorithm initialises the buffer of signal s with one buffer element (line 1), initialises the buffering schedule and lifetime to be empty (lines 2 and 3), initialises the `prev` pointers of the writers to reference the initial buffer element (line 4), and initialises the `next` pointers of the writers to be unknown (line 5). Line 6 initialises *current*, which tracks the buffer element that the readers access (*prev*) and the writer instance that wrote the element's value (t_w^i), to reference the initial buffer element and to a currently unknown writer instance.

Algorithm 1 processes the uses of signal s in chronological order (lines 7–9) by making buffering decisions whenever a task ends or starts its LET (beginning at lines 10, 12, 15, and 26). When a reader ends its LET (line 10), it no longer needs to access its allocated buffer element. Thus, the reader's instance is removed from its allocated buffer element (line 11). When a writer ends its LET (line 12), its *next* pointer is assigned to its *prev* pointer, and its *next* pointer is reset to unknown (line 13). If the signal's value is defined by the writer's instance (line 14), then *current* is updated with the writer's buffer element and task instance.

When a reader starts its LET (line 15), it reads from the current buffer element, i.e., *current.prev*. This is recorded in the buffering schedule (line 16). Additionally, the reader's instance occupies the current buffer element, e_{prev}, referenced by *current.prev* (line 17). Because the current buffer element now has a writer and a corresponding reader, the time between the writer's LET start and the reader's LET end contributes towards the buffer's lifetime. Line 19 calculates the latest LET end times among the readers (*maxLetEnd*). If no writer instance has replaced the buffer's initial value (line 20), then the initial buffer element is used and the interval $\langle 0, maxLetEnd \rangle$ is added to the lifetime. For example, at time 0 ms in Figure 9, the interval $\langle 0\ ms, 4\ ms \rangle$ is added. Otherwise, the interval $\langle current.t_a^i.letStart, maxLetEnd \rangle$ is added (line 23). For example, at time 1.2 ms in Figure 9, task t_1 is the only reader to start and its LET ends at 2.2 ms, so the interval $\langle 0\ ms, 2.2\ ms \rangle$ is added. A more compact representation of intervals can be used [GKKL16] to simplify subsequent interval calculations.

The remainder of the SBP pseudocode is presented in Algorithm 2. When a writer starts its LET (line 26), an available buffer element needs to be found. Line 27 finds all occupied buffer elements, which are those being used by the writers (*Next*), those being used by the readers, and the buffer element *current.prev* that will be used by future readers that start during the LET of the writers in *writersStart*. For example, when t_0's second instance starts at 1 ms in Figure 9, *current.prev* will be used at 1.2 ms by t_1's second instance. For each writer instance (line 28), its previous buffer element is reused if available (line 30), otherwise an available buffer element is used (line 33). If all elements in the buffer are occupied (line 31), then the buffer is extended with a new element (line 35) that is allocated to the writer instance (line 36). This is recorded in the buffering schedule (line 38). Afterwards, the set of occupied buffer elements is updated (line 40) and the writer instance contributes towards the buffer's lifetime (line 41).

After all uses of signal s have been processed, the buffer's lifetime is updated to include

Algorithm 1 Part I of Sbp. Returns the buffering schedule and lifetime for signal s.

Input: $uses_s$ (write and read uses of signal s), $selectedWI_s$ (selected writer instances for signal s), and hp (hyper-period duration)

Output: $buffSch_s$ (buffering schedule of signal s), and $lifetime_s$ (buffer's lifetime)

1: $buff_s \leftarrow \{e_0 \leftarrow \emptyset\}$ // Buffer has one available element.
2: $buffSch_s \leftarrow \emptyset$ // Empty buffering schedule for signal s.
3: $lifetime_s \leftarrow \emptyset$ // Empty set of intervals for the buffer's lifetime.
4: $Prev \leftarrow \{prev_{t_w} \leftarrow e_0 \mid t_w \in \mathrm{W}(s)\}$ // `prev` of all writers is element e_0.
5: $Next \leftarrow \{next_{t_w} \leftarrow null \mid t_w \in \mathrm{W}(s)\}$ // `next` of all writers is unknown.
6: $current \leftarrow \langle prev \leftarrow e_0, t_w^i \leftarrow null \rangle$ // Tracks the current `prev` pointer.

7: **while** $uses_s \neq \emptyset$ **do**
    // Get all signal usages with the earliest timestamp.
8: $earliestUses \leftarrow \mathrm{GetEarliestUses}(uses_s)$
9: $uses_s \leftarrow uses_s \setminus earliestUses$

    // Reader ends its LET: no longer needs its buffer element.
10: $readersEnd \leftarrow \{use.t_r^i \mid use.\phi = \mathcal{R}^{end}, use \in earliestUses\}$
11: $buff_s \leftarrow \{e_i \setminus readersEnd \mid e_i \in buff_s\}$

    // Writer ends its LET: its value can now be read.
12: $writersEnd \leftarrow \{use.t_w^i \mid use.\phi = \mathcal{W}^{end}, use \in earliestUses\}$
13: $\forall t_w^i \in writersEnd : prev_{t_w} \leftarrow next_{t_w}$ **and** $next_{t_w} \leftarrow null$
14: **if** $\exists t_w^i \in writersEnd \cap selectedWI_s$ **then** $current \leftarrow \langle prev_{t_w}, t_w^i \rangle$ **end if**

    // Reader starts its LET: occupies the writer's `prev` element.
15: $readersStart \leftarrow \{use.t_r^i \mid use.\phi = \mathcal{R}^{start}, use \in earliestUses\}$
16: $buffSch_s \leftarrow buffSch_s \cup \{\langle t_r^i, current.prev \rangle \mid t_r^i \in readersStart\}$ // Record allocation.
17: $current.prev$ **as** e_{prev} : $e_{prev} \leftarrow e_{prev} \cup readersStart$

    // Update the buffer's lifetime whenever it is used by a reader.
18: **if** $readersStart \neq \emptyset$ **then**
19: $maxLetEnd \leftarrow \max\left(\{t_r^i.letEnd \mid t_r^i \in readersStart\}\right)$
20: **if** $current.t_a^i = null$ **then** // First writer instance has not finished.
21: $lifetime_s \leftarrow lifetime_s \cup \{\langle 0, maxLetEnd \rangle\}$ // Initial value is needed.
22: **else**
23: $lifetime_s \leftarrow lifetime_s \cup \{\langle current.t_a^i.letStart, maxLetEnd \rangle\}$
24: **end if**
25: **end if**

// Continued...

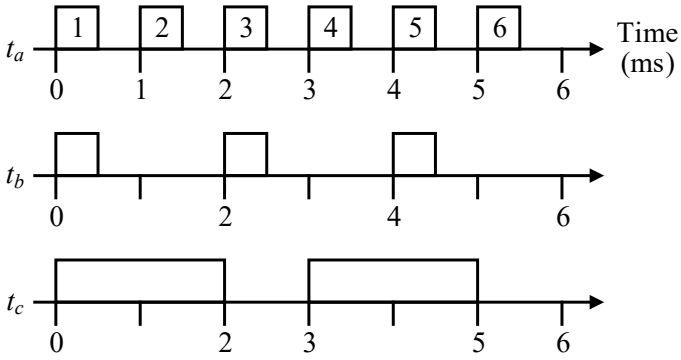
Algorithm 2 Part II of Sbp.

// Writer starts its LET: find an available buffer element to occupy.
26: $writersStart \leftarrow \{use.t^i_w \mid use.\phi = \mathcal{W}^{start}, use \in earliestUses\}$
27: $occupiedBuff \leftarrow Next \cup \{e \in buff_s \mid e \neq \emptyset\}$
$\cup \{current.prev \mid \text{GetUses}(uses_s, t^i_w.letStart, t^i_w.letEnd, \mathcal{R}^{start}) \neq \emptyset, t^i_w \in writersStart\}$
28: **for** $t^i_w \in writersStart$ **do**
29: **if** $prev_{t_w} \notin occupiedBuff$, **where** $prev_{t_w} \in Prev$ **and** $next_{t_w} \in Next$ **then**
30: $next_{t_w} \leftarrow prev_{t_w}$ // Keep using its allocated buffer element.
31: **else**
32: **if** $\exists e \in buff_s \setminus occupiedBuff$ **then**
33: $next_{t_w} \leftarrow e$ // Allocate an available buffer element.
34: **else**
35: $buff_s \leftarrow buff_s \cup \{e \leftarrow \emptyset\}$ // Create and allocate to a new buffer element.
36: $next_{t_w} \leftarrow e$
37: **end if**
38: $buffSch_s \leftarrow buffSch_s \cup \{\langle t^i_w, next_{t_w}\rangle\}$ // Record change in allocation.
39: **end if**
40: $occupiedBuff \leftarrow occupiedBuff \cup \{next_{t_w}\}$
// Writer instance contributes to the buffer's lifetime.
41: $lifetime_s \leftarrow lifetime_s \cup \{\langle t^i_w.letStart, t^i_w.letEnd\rangle\}$
42: **end for**
43: **end while**

// The last writer instance writes the next hyper-period's initial value.
44: $lifetime_s \leftarrow lifetime_s \cup \{\langle current.t^i_w.letStart, hp\rangle\}$

45: **if** $current.prev \neq e_0$ **then**
46: Remember to copy the value in $current.prev$ to e_0 when the hyper-period ends
47: **end if**

// Reduce the buffering schedule: when the same buffer element is allocated to
// consecutive instances of a reader task, only the first allocation is necessary.
48: **for** $\langle t^i_r, e_x\rangle, \langle t^k_r, e_x\rangle \in buffSch_s$ **where** $t_r \in \mathrm{R}(s)$, **such that** $\nexists \langle t^j_r, e_y\rangle$ **where** $i < j < k$ **do**
49: $buffSch_s \leftarrow buffSch_s \setminus \{\langle t^k_r, e_x\rangle\}$
50: **end for**



(a) Logical task schedule for writer t_a and readers t_b and t_c. The values read by t_b and t_c are omitted.

$t_a \xrightarrow{s,\ 3ms} t_b$

$t_a \xrightarrow{s,\ 3ms} t_c$

(b) Data age constraints.

Figure 10: Small example to illustrate the suppression of unnecessary writes.

the use of the buffer to hold the signal's last value until the next hyper-period (line 44). As a side note (lines 45–47), if the signal's last value is not written into the buffer's initial element, then it must be copied into the initial element when the hyper-period ends. Finally, the buffering schedule can be reduced by discarding allocations where the same buffer element is allocated to consecutive instances of a reader (lines 48–50).

The buffer memory complexity of SBP is never worse than that of DBP, because further optimisations are possible (see Sections 7.2 and 7.3). Buffer bounds can be computed exactly for SBP by inspecting the number of buffer elements in each buffering schedule. The auxiliary memory for storing the buffer indexes needed by a task, e.g., in a lookup table, is linear to the length of the hyper-period. The time complexity of SBP at design time is linear to the length of the hyper-period when a constant-time algorithm is used to find free buffer elements. However, the time complexity increases when additional buffer optimisations are employed. At run-time, the time complexity of SBP is constant.

7.2 Suppression of Unnecessary Writes

A buffering schedule can be improved by analysing whether the output of a writer's instance is actually needed by a reader. Our analysis considers two possibilities: an output is unnecessary if (1) it is always overwritten before it is read by any task, e.g., when a signal is being under-sampled, or (2) a signal's data age constraint permits tasks to read an older value instead of a fresher one. The second possibility affects the design's data-flow and the designer must decide whether this is acceptable. Note that none of the writes to signals with the global programming style can be suppressed; it is only possible for the local programming style.

Figure 10a is a small logical task schedule for a writer t_a and some readers t_b and t_b, communicating over signal s. Notice that the output from the first and fifth instances of t_a, i.e., t_a^0 and t_a^4, can be suppressed because they are overwritten by t_a^1 and t_a^5 before they can even be read. The data age constraints specified in Figure 10b allow t_b and t_c to read values of s that are up to 3 ms old. For example, the second instance of t_b could read the signal's initial value, or the output from t_a's first or second instances. Thus, the latest instance that t_b can read from t_a is t_a^1. Table 5 gives the writer instances that each reader instance could potentially read from. The reading of the signal's initial value is represented as instance -1 of the writer.

The goal now is to find a subset of writer instances that satisfies all reader instances, with preference for the latest possible writers. This is a variation of the NP-hard *set cover*

Algorithm 3 GetWriterUses returns the necessary writer instances of a signal.

Input: s (signal of interest), $selectedWI_s$ (selected writer instances for s), $allRI_s$ (reader instances of s), and $DataAges$ (data age constraints)
Output: $uses_s$ (uses of signal s)

$potentialTable_s \leftarrow \emptyset$ // Empty table of potential reads.
$latestTable_s \leftarrow \emptyset$ // Empty table of latest writer instances to read.
$uses_s \leftarrow \emptyset$ // Empty set of write usages.
$satRI_s \leftarrow \emptyset$ // Empty set of satisfied reader instances.

// Make a table of potential reads for each writer instance (including the initial value).
for $t_r^i \in allRI_s$ **do**
  for $t_w^j \in selectedWI_s \cup \{t_w^{-1} = \langle 0, 0, 0 \rangle\}$ **do**
    if $0 \leq (t_r^i.letStart - t_w^j.letEnd) \leq \delta$ **where** $t_w \xrightarrow{s,\ \delta} t_r \in DataAges$ **then**
      $potentialTable_s(t_w^j) \leftarrow potentialTable_s(t_w^j) \cup \{t_r^i\}$
    end if
  end for
end for
Sort $potentialTable_s$ such that $t_w^j.letEnd$ of row n is earlier than $t_{w'}^{j'}.letEnd$ of row $n+1$

// Find the latest possible writer instance that each reader instance can read from.
for consecutive rows t_w^j, t_w^k in $potentialTable_s$ in sorted order **do**
  $latestTable_s(t_w^j) \leftarrow potentialTable_s(t_w^j) \setminus potentialTable_s(t_w^k)$
  $l \leftarrow k$ // Record the last row of the potential table.
end for
$latestTable_s(t_w^l) \leftarrow potentialTable_s(t_w^l)$
Sort $latestTable_s$ such that $t_w^j.letEnd$ of row n is earlier than $t_{w'}^{j'}.letEnd$ of row $n+1$

// Record the writer instances that are necessary for the reader instances.
for each row t_w^j in $latestTable_s$ in sorted order **do**
  if $latestTable_s(t_w^j) \not\subseteq satRI_s$ **then**
    if $j \neq -1$ **then** // Initial value is always available.
      $uses_s \leftarrow uses_s \cup \{\langle t_w^j, t_w^j.letStart, \mathcal{W}^{start} \rangle, \langle t_w^j, t_w^j.letEnd, \mathcal{W}^{end} \rangle\}$
    end if
    $satRI_s \leftarrow satRI_s \cup potentialTable_s(t_w^j)$
  end if
end for

// Last writer instance may be needed to initialise the signal's value
// in the next hyper-period.
$uses_s \leftarrow uses_s \cup \{\langle t_w^l, t_w^l.letStart, \mathcal{W}^{start} \rangle, \langle t_w^l, t_w^l.letEnd, \mathcal{W}^{end} \rangle\}$

Table 5: The writer instances that satisfy each reader instance from Figure 10 (a signal's initial value is represented as instance -1 of the writer)

Reader instance	**Writer t_a's instance number**						
	-1	**0**	**1**	**2**	**3**	**4**	**5**
t_b^0	✓						
t_b^1	✓	✓	✓				
t_b^2			✓	✓	✓		
t_c^0	✓						
t_c^1		✓	✓	✓			

Table 6: Potential table for Figure 10

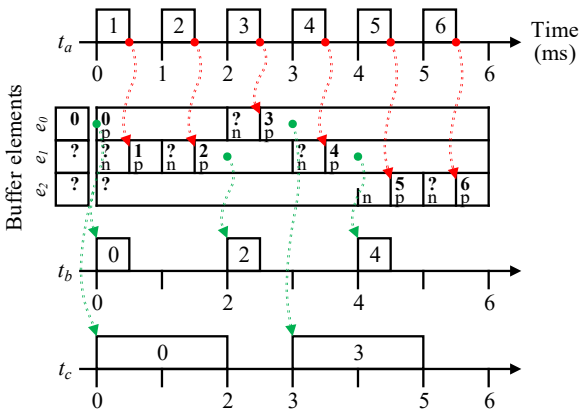
Writer instance	**Reader instances**
t_a^{-1}	$\{t_b^0, t_b^1, t_c^0\}$
t_a^0	$\{t_b^1, t_c^1\}$
t_a^1	$\{t_b^1, t_b^2, t_c^1\}$
t_a^2	$\{t_b^2, t_c^1\}$
t_a^3	$\{t_b^2\}$
t_a^4	$\{\ \}$
t_a^5	$\{\ \}$

Table 7: Latest table for Figure 10

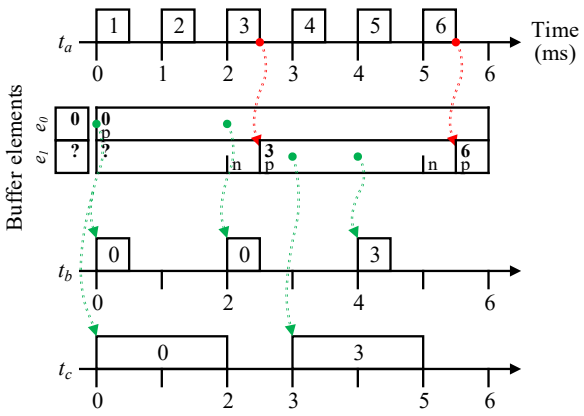
Writer instance	**Reader instances**
t_a^{-1}	$\{t_b^0, t_c^0\}$
t_a^0	$\{\ \}$
t_a^1	$\{t_b^1\}$
t_a^2	$\{t_c^1\}$
t_a^3	$\{t_b^2\}$
t_a^4	$\{\ \}$
t_a^5	$\{\ \}$

Table 8: Necessary writer instances for Figure 10

Selected writer instances	**Satisfied reader instances**
$\{t_a^{-1}\}$	$\{t_b^0, t_b^1, t_c^0\}$
$\{t_a^{-1}\}$	$\{t_b^0, t_b^1, t_c^0\}$
$\{t_a^{-1}\}$	$\{t_b^0, t_b^1, t_c^0\}$
$\{t_a^{-1}, t_a^2\}$	$\{t_b^0, t_b^1, t_b^2, t_c^0, t_c^1\}$



(a) No writes suppressed.



(b) Unnecessary writes suppressed. One less buffer element is needed.

Figure 11: Applying SBP on the logical task schedule of Figure 10a.

problem [Kar72]. Algorithm 3 defines our heuristic for solving this problem. Starting from lines 5–12, a compact version of Table 5 is constructed, called a *potential table*, shown in Table 6. Each row of the potential table is index by the writer's instance t_w^j, and $potentialTable_s(t_w^j)$ returns t_w^j's set of reader instances. From this, lines 13–18 finds the latest (freshest) possible writer instance that each reader instance can use, producing a *latest table* shown in Table 7. The writer instances in this table with a non-empty set of readers are sufficient to satisfy all reader instances, e.g., $\{t_a^{-1}, t_a^1, t_a^2, t_a^3\}$. To refine these writer instances to the necessary ones, lines 19–26 iterate through the latest table to select the earliest writer instance with a non-empty set of reader instances, e.g., t_a^{-1}. The selection is saved by storing the writer's use of the the signal into $uses_s$ (line 22). The selected writer instance is likely to satisfy additional reader instances, i.e., those in the potential table. For example, t_a^{-1} also satisfies t_b^1 and t_c^0. Thus, a set of satisfied reader instances, $satRI_s$, is maintained. Continuing through the latest table, we select the next writer instance, e.g., t_a^2, that has a reader instance that is not already satisfied, and update $uses_s$ and $satRI_s$ accordingly. This continues until all reader instances are satisfied. The set of necessary writer instances for our example is $\{t_a^{-1}, t_a^2\}$, and the iterative selection of necessary writer instances is shown in Table 8. If the output of the last writer instance is needed to initialise the signal's value in next hyper-period, then that writer instance has to be included, e.g., $\{t_a^{-1}, t_a^2, t_a^5\}$.

Figure 11a illustrates the buffering schedule that is constructed when Sbp (Algorithms 1 and 2) is applied without the suppression of unnecessary writes. Contrast this to Figure 11b, where unnecessary writes are suppressed by taking into account the data age constraints of Figure 10b. Because only two writer instances need to write to the buffer, one less buffer element is needed to preserve the LET semantics.

7.3 Constructing the SBP Buffering Schedules

The construction of a buffering schedule for a signal is detailed in Algorithm 4. For each signal, it identifies all time intervals that the signal could be used by its readers (lines 2–3) and writers (lines 4–9). At the design phase, these intervals are equal to the task instances' LETs. The LET start and end times of all reader instances are recorded as $\mathcal{R}^{start}$ and $\mathcal{R}^{end}$, respectively. When a signal has the global programming style (line 5), where intermediate signal values are written directly to the buffer, the LET start and end times of all writer instances are recorded as $\mathcal{W}^{start}$ and $\mathcal{W}^{end}$, respectively. For the local programming style (lines 7–8), where only final signal values are written to the buffer, only the $\mathcal{W}^{start}$ and $\mathcal{W}^{end}$ of the necessary writer instances identified by GetWriterUses (Algorithm 3) are recorded. Based on the chronological order of the recorded time points, the static buffering protocol Sbp (Algorithms 1 and 2) decides which buffer element each task accesses, and the total number of buffer elements needed to properly preserve the LET semantics (lines 10–12). The decisions are recorded as a buffering schedule ($buffSch_s \in BuffSchs$), and the lifetime ($lifetime_s \in Lifetimes$) of the buffer is computed. Two signal buffers have disjoint lifetimes if none of their intervals overlap. Thus, once all signals have a buffering schedule, pairs of buffers with disjoint lifetimes are found (lines 14–19), and this information is used during deployment to selectively merging buffers and save memory (Figure 8, Step 5). More efficient interval calculations can be found in literature; see, e.g., Gavryushkin et al. [GKKL16].

Algorithm 4 SbpBufferingSchedules returns the buffering schedules of all signals, and pairs of signal buffers with disjoint lifetimes.

Input: S (all signals), hp (hyper-period duration), $DataAges$ (data age constraints of all signals), $AllWI$(all writer instances of all signals), and $SelectedWI$(writer instances selected from $AllWI$)

Output: $BuffSchs$ (buffering schedules of all signals), and $DisjBuffs$ (pairs of signals with disjoint buffer lifetimes)

1: **for** $s \in S$ **do**
   // All reader instances will use signal s.
2: $allRI_s \leftarrow \{t_r^i \mid 0 \leq i < \frac{hp}{t_r.period}, i \in \mathbb{N}, t_r \in \mathrm{R}(s)\}$
3: $uses_s \leftarrow \{\langle t_r^i, t_r^i.letStart, \mathcal{R}^{start}\rangle, \langle t_r^i, t_r^i.letEnd, \mathcal{R}^{end}\rangle \mid t_r^i \in allRI_s\}$

4: **if** $s.style = global$ **then** // All writer instances use signal s.
5: $uses_s \leftarrow uses_s \cup \{\langle t_w^i, t_w^i.letStart, \mathcal{W}^{start}\rangle, \langle t_w^i, t_w^i.letEnd, \mathcal{W}^{end}\rangle \mid t_w^i \in allWI_s\}$
6: **else** // Not all writer instances may need to use signal s.
7: $uses_s \leftarrow uses_s \cup \mathrm{GetWriterUses}(s, selectedWI_s, allRI_s, DataAges)$
8: $selectedWI_s \leftarrow \{t_w^i \mid \langle t_w^i, \sigma, \phi\rangle \in uses_s\}$
9: **end if**

10: $\langle buffSch_s, lifetime_s\rangle \leftarrow \mathrm{Sbp}(uses_s, selectedWI_s, hp)$
11: $BuffSchs \leftarrow BuffSchs \cup \{buffSch_s\}$
12: $Lifetimes \leftarrow Lifetimes \cup \{lifetime_s\}$
13: **end for**

   // Identify pairs of signal buffers with disjoint lifetimes.
14: $DisjBuffs \leftarrow \emptyset$
15: **for** $lifetime_{s_0}, lifetime_{s_1} \in Lifetimes$ **where** $lifetime_{s_0} \neq lifetime_{s_1}$ **do**
16: **if** $\nexists interval_{s_0} \in lifetime_{s_0}$ **and** $\nexists interval_{s_1} \in lifetime_{s_1}$,
   where $interval_{s_0}.start \leq interval_{s_1}.start \leq interval_{s_0}.end$
   or $interval_{s_1}.start \leq interval_{s_0}.start \leq interval_{s_1}.end$
   then
17: $DisjBuffs \leftarrow DisjBuffs \cup \{\langle s_0, s_1\rangle\}$ // No intervals of s_0 and s_1 overlap.
18: **end if**
19: **end for**

7.4 Discussion

By virtue of static scheduling over a hyper-period, more opportunities are available for reducing SBP's buffer memory requirement as compared to DBP. Unlike DBP, SBP uses data age constraints to suppress unnecessary writes to the signal buffers, thereby reducing bus traffic to global memory. Although the worst-case run-time cost of DBP is already constant, the elimination of redundant allocations from the buffering schedule means that only some task instances incur the overhead of updating their buffer index. However, this requires the variables storing the current buffer indexes to be on the heap in order to persist across task instances. Note that redundant allocations for the writers are eliminated when they reuse their previously allocated buffer element. Exact memory sizes are computed from the static buffering schedules, which helps one to mitigate the over-provisioning of memory resources. During the deployment phase, the buffering schedule of each signal can be used to find and merge buffers that have disjoint lifetimes. The design phase optimisations presented in this section apply also to LET tasks with irregular periods. This is because only the LET start and end times of each task instance are necessary.

8 Deployment Phase Optimisations

The deployment phase optimisations are platform dependent, and decide on the task-to-core allocations, the selection of buffering protocol for each signal, and the signal buffer-to-memory module allocations. The optimisation goal is to minimise task execution times, subject to constraints on SBP and PTP buffering selection, task and signal allocations, task schedulability, memory module capacities.

To properly model the overheads of PTP and SBP buffering, Section 8.1 briefly describes how LET tasks can be implemented and scheduled under AUTOSAR OS in a semantics preserving manner. The system model (see Section 5) is then extended with additional task scheduling parameters (see Section 8.2), necessary for formulating the proposed MILP optimisation problem (see Section 8.3). A corresponding genetic algorithm is presented, which provides possibly suboptimal results but at faster solving times (see Section 8.4). Physical task schedules are then constructed for each core from the optimisation solution, as well as reducing system response times by scaling down the task timing parameters (see Section 8.5). Because more information is known about task execution times and buffer allocations, the SBP buffering schedules can be refined further and signal buffers can be merged together (see Sections 8.6–8.7).

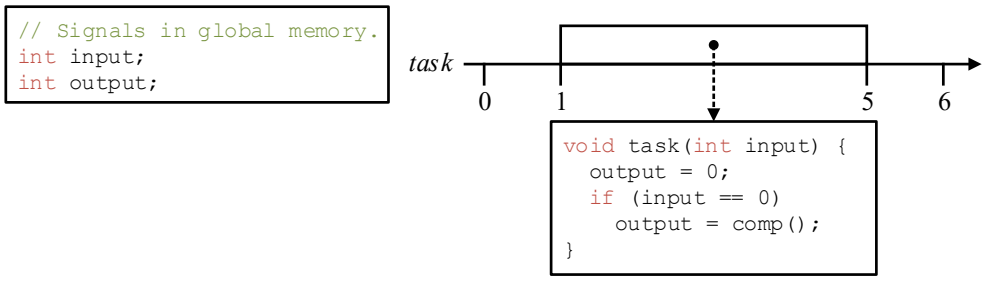
8.1 Realisation of LET Tasks under AUTOSAR

Section 2.5 has already discussed the use of so-called schedule tables to implement task schedules over one hyper-period. To understand the deployment optimisations, it is necessary to discuss how the PTP and SBP buffering protocols can be implemented in a semantics preserving manner using schedule tables. For PTP, the LET tasks must read signals into their local buffers when they start their LETs, and must write out their local buffers when they end their LETs. This is achieved by generating a pair of *LET start* and *LET end* routines for each LET task, which contain the necessary code to perform the required buffering actions. Figure 12a is a simple example of a LET task, and its implementation with PTP is illustrated in Figure 12b. The LET task now has LET start and end routines for managing the PTP buffers, and has a computation task that is the original LET task with buffered signal accesses. The LET start and end routines are scheduling as tasks that execute at the defined LET start and end times. The computation task is scheduled to execute between the start and end routines. If multiple LET tasks start or end their LETs at the same time, then the LET end routines must be scheduled to execute before any of the LET start routines. It is also important to align the LET start and end routines across the cores to ensure that the LET semantics are preserved. However, it is beyond the scope of this report to discuss the technical details of the scheduling algorithm.

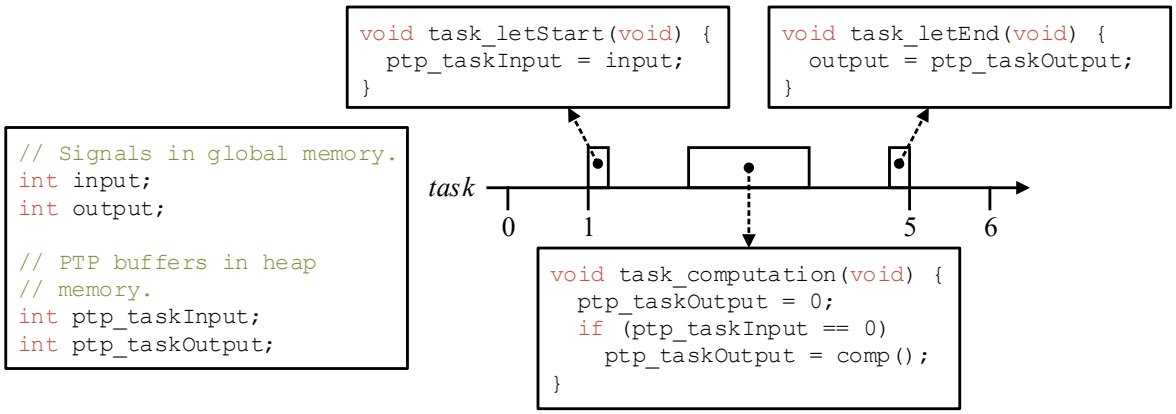
For SBP, there is no need to create LET start and end routines, because the required buffer element that each task instance needs to access for semantics preserving communication is statically known. The signal is converted into an array in memory, with each buffer element mapped to an array (buffer) index. Thus, a LET task only needs to update its required buffer indexes before executing its computations. Signal accesses in the task's computation are replaced by accesses to the buffer array. The modified task can be scheduled for execution at any time within its LET. For the original LET task in Figure 12a, its implementation with SBP is illustrated in Figure 12c.

8.2 System Model Extensions

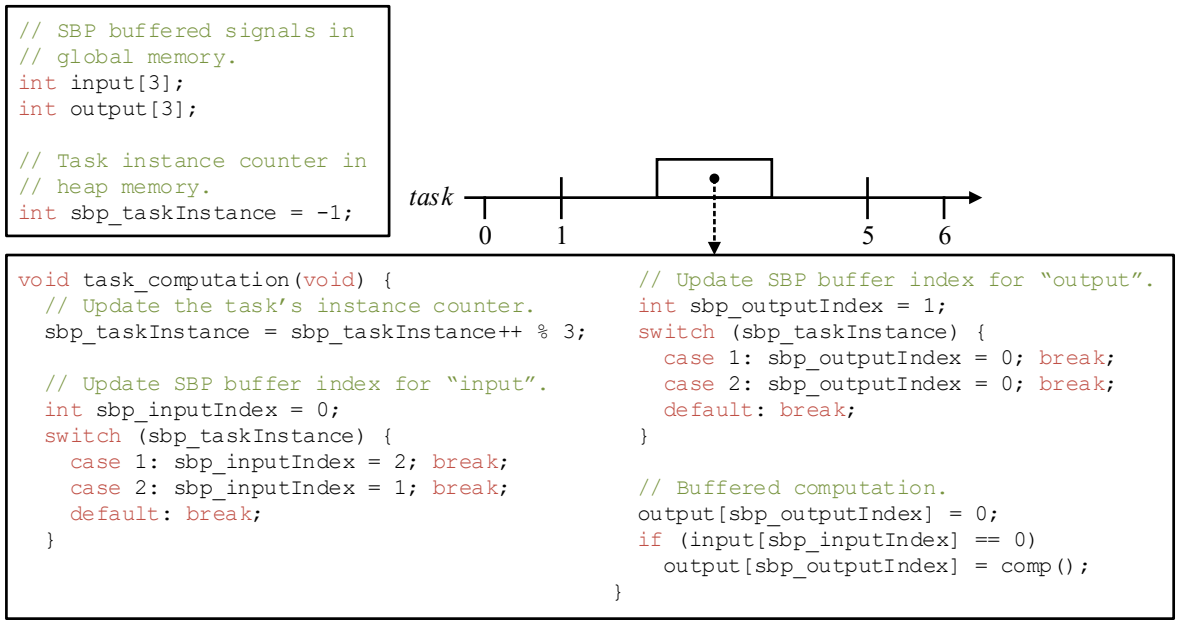
We extend the system model described in Section 5 with additional task scheduling information. Each core's task schedule is divided into an identical sequence of uniquely identifiable



(a) Original LET task. Global programming style used for the input and output signals.



(b) Implementation with PTP. The buffers are allocated on the heap so that their values persist across the LET start/end routines and the task's computations.



(c) Implementation with SBP. The signals are transformed into arrays, and a counter is used to track the task's instance. Assume that the task executes three instances over one hyper-period, and that the input and output signals each need three buffer elements. For the input signal, assume that indexes 0, 2, and 1 are accessed in the task's first, second, and third instances, respectively. For the output signal, assume that index 1 is accessed in the task's first instance, and that index 0 is accessed in the second and third instances. If signal indexes are also allocated to the heap, then only the assignment of new index values is needed.

Figure 12: Realising a LET task with PTP or SBP buffering in AUTOSAR.

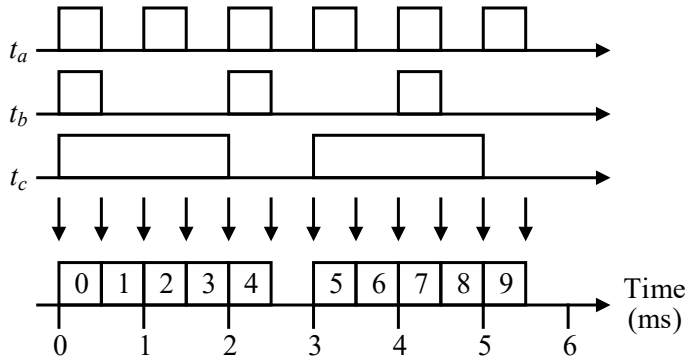


Figure 13: Scheduling slots for the logical task schedule in Figure 10a. Only the slots that coincide with the LET tasks are of interest and are labelled from 0 to 9.

slots over the hyper-period with duration d, i.e., $slot = \langle d \rangle \in Slots$. This is achieved by projecting the logical task schedule onto a single timeline and dividing the LETs at every LET boundary, as depicted in Figure 13. In each slot, execution time can be allocated to several tasks. The LET of a task instance can also be represented as a set of scheduling slots, i.e., $slots = \{slot \in Slots\}$. The first instance of t_c in Figure 13 would have $slots = \{0, 1, 2, 3\}$. Because tasks can be scheduled preemptively, the task schedules must accommodate the context-switching overhead. The maximum number of possible context-switches, $cs \in \mathbb{N}$, during a task's computation may be estimated from the number of scheduling slots needed to represent its instances, e.g., $cs = \max\left(\left\{|t_a^i.slots| \text{ where } 0 \leq i < \frac{hp}{t_a.period}\right\}\right)$. The definitions for a task instance and task are updated to $t_a^i = \langle periodStart, letStart, letEnd, slots \rangle$ and $t_a = \langle period, letStart, letEnd, acc, instr, cs \rangle$, respectively.

8.3 Mixed-Integer Linear Programming (MILP) Formulation

We use MILP [BGG+71] to find task-to-core and buffer-to-memory module allocations, and to select a buffering protocol for each signal for a LET-based AUTOSAR design. The objective is to minimise the total task execution time that needs to be allocated in the physical task schedules:

$$\text{Minimise}: \sum_{t_a \in T} \sum_{slot \in t_a^i.slots} slot.t_a^i.alloc \tag{4}$$

where $slot.t_a^i.alloc$ is a real variable for the execution time that is allocated to task instance t_a^i in the scheduling $slot$. This objective is subject to constraints on SBP and PTP buffering selection, memory module capacities, task and signal allocations, and task schedulability, which are detailed below. Table 9 summarises the variables and constants used in the MILP formulation. We assume that all memory sizes and timing information are expressed in consistent units, e.g., as bits and in milliseconds, respectively.

SBP and PTP buffering. The Boolean variables sbp_s and ptp_s select whether SBP or PTP is used, respectively, for signal s:

$$\forall s \in S : sbp_s + ptp_s = 1 \tag{5}$$

When SBP is used, the Boolean variable $sbp_s.m$ is 1 *iff* the SBP buffer of signal s is allocated to memory module m. The buffer can only be allocated to one memory module:

$$\forall s \in S : sbp_s = \sum_{m \in M} sbp_s.m \tag{6}$$

Table 9: Summary of the variables and constants used in the MILP formulation

Type	Variable	Description
Boolean	sbp_s, ptp_s	Selection of buffering protocol for signal s.
	$sbp_s.m$, $ptp_s.m$	Allocation of signal s to memory module m.
	$ptp_s.t_a.m$	Allocation of task t_a's local PTP buffer for signal s to memory module m.
	$t_a.c$	Allocation of task t_a to core c.
	$t_a.c.sbp_s$, $t_a.c.sbp_s.m$, $t_a.c.ptp_s.m$, $t_a.c.ptp_s.t_a.m$	Variables needed to linearise the relationships between task, core, buffer, and memory allocations.
Integer	$m.sbp$, $m.ptp$	Total SBP and PTP memory requirement for memory module m.
Real	$t_a.wcet_b$, $t_a.wcet_c$	Worst-case execution time for task t_a's buffer management and computations (instructions and signal accesses).
	$slot.t_a^i.alloc$, $t_a^i.c.slot.alloc$	Execution time allocated to task instance t_a^i in the scheduling $slot$, and its linearised form with the task's core allocation.

Type	Constant	Description
Integer	$s.size$, $s.n$, $s.w$	Signal information on data size, number of SBP buffer elements, number of pathway transfers needed per access.
	$m.size$	Capacity of memory module m.
	$t_a.acc$, $t_a.cs$	Number of signal accesses by task t_a, and number of context-switches during task t_a's execution.
Real	$t_a.letStart$, $t_a.letEnd$, $t_a.instr.c.wcet$	Task t_a's LET start and end times, and linearisation of its instruction's worst-case execution time on core c.
	$wcet_{cs}.c$, $wcet_{sbp}.c$, $wcet_{ptp}.c$	Core specific overheads for a task to context-switch and to manage one SBP or PTP buffer element.
	$path.c.m$	Latency of the pathway between core c and memory module m.
	$slot.d$	Duration of scheduling $slot$.

For each memory module m, the (positive) integer variable $m.sbp$ tracks the total memory needed for its allocated SBP buffers. The total size of each SBP buffer is $s.size \times s.n$:

$$\forall m \in M : m.sbp = \sum_{s \in S} sbp_s.m \times s.size \times s.n \tag{7}$$

When PTP is used, the Boolean variable $ptp_s.t_a.m$ is 1 *iff* task t_a's PTP buffer for signal s is allocated to memory module m. The Boolean variable $ptp_s.m$ is 1 if signal s itself is allocated to memory module m. Each PTP buffer and corresponding signal can only be allocated to one memory module:

$$\forall t_a \in T \; \forall \langle s, n \rangle \in t_a.acc : ptp_s = \sum_{m \in M} ptp_s.t_a.m \tag{8}$$

$$\forall s \in S : ptp_s = \sum_{m \in M} ptp_s.m \tag{9}$$

For each memory module m, the (positive) integer variable $m.ptp$ tracks the total memory needed for its allocated PTP buffers and corresponding signals. The size of each PTP buffer and signal is $s.size$:

$$\forall m \in M : m.ptp = \sum_{t_a \in T} \sum_{\langle s,n \rangle \in t_a.acc} ptp_s.t_a.m \times s.size + \sum_{s \in S} ptp_s.m \times s.size \tag{10}$$

Finally, for each memory module m, the total memory available for its allocated SBP and PTP buffers is limited by its capacity:

$$\forall m \in M : m.sbp + m.ptp \leq m.size \tag{11}$$

LET tasks. The Boolean variable $t_a.c$ is 1 *iff* task t_a is allocated to core c. A task can only be allocated to one core:

$$\forall t_a \in T : \sum_{c \in C} t_a.c = 1 \tag{12}$$

The following Boolean variables are used to linearise the relationships between task, core, buffer, and memory allocations:

- $t_a.c.sbp_s$ is 1 *iff* task t_a is on core c and signal s uses SBP (linearisation of $t_a.c \times sbp_s$).
- $t_a.c.sbp_s.m$ is 1 *iff* task t_a is on core c and the SBP buffer of signal s is on memory module m (linearisation of $t_a.c \times sbp_s.m$).
- $t_a.c.ptp_s.t_a.m$ is 1 *iff* task t_a is on core c and its PTP buffer of signal s is on memory module m (linearisation of $t_a.c \times ptp_s.t_a.m$).
- $t_a.c.ptp_s.m$ is 1 *iff* task t_a is on core c and signal s itself is on memory module m (linearisation of $t_a.c \times ptp_s.m$).

The linearised Boolean relationships are:

$$0 \leq t_a.c + sbp_s - 2 \times t_a.c.sbp_s \leq 1 \tag{13}$$

$$0 \leq t_a.c + sbp_s.m - 2 \times t_a.c.sbp_s.m \leq 1 \tag{14}$$

$$0 \leq t_a.c + ptp_s.t_a.m - 2 \times t_a.c.ptp_s.t_a.m \leq 1 \tag{15}$$

$$0 \leq t_a.c + ptp_s.m - 2 \times t_a.c.ptp_s.m \leq 1 \tag{16}$$

The worst-case time for task t_a to manage its buffers is tracked by the (positive) real variable $t_a.wcet_b$. With respect to t_a's allocated core c, the variable captures the total time for t_a to set its SBP buffer indexes ($wcet_{sbp}.c$), to context-switch to t_a's LET start and end routines for PTP buffering ($wcet_{cs}.c$), and to copy signals to and from t_a's PTP buffers:

$$\begin{aligned}\forall t_a \in T : t_a.wcet_b = &\sum_{c \in C} \sum_{\langle s,n \rangle \in t_a.acc} \left(t_a.c.sbp_s \times wcet_{sbp}.c\right) + 2 \times \sum_{c \in C} t_a.c \times wcet_{cs}.c \\ &+ \sum_{c \in C} \sum_{m \in M} \sum_{\langle s,n \rangle \in t_a.acc} t_a.c.ptp_s.t_a.m \times \left(wcet_{ptp}.c + path.c.m \times s.w\right) \\ &+ \sum_{c \in C} \sum_{m \in M} \sum_{\langle s,n \rangle \in t_a.acc} t_a.c.ptp_s.m \times path.c.m \times s.w\end{aligned} \tag{17}$$

The worst-case time for task t_a to execute its computations is tracked by the (positive) real variable $t_a.wcet_c$. With respect to t_a's allocated core c, the variable captures the total time t_a needs to execute its instructions ($t_a.instr.c.wcet$), to context-switch to and from t_a during preemptive scheduling ($wcet_{cs}.c$), and for t_a to access its buffered signals. Note that n, from $\langle s,n \rangle \in t_a.acc$, is the number of times that t_a accesses signal s during its execution:

$$\begin{aligned}\forall t_a \in T : t_a.wcet_c = &\sum_{c \in C} t_a.c \times t_a.instr.c.wcet + \sum_{c \in C} t_a.c \times wcet_{cs}.c \times t_a.cs \\ &+ \sum_{c \in C} \sum_{m \in M} \sum_{\langle s,n \rangle \in t_a.acc} t_a.c.sbp_s.m \times path.c.m \times s.w \times n \\ &+ \sum_{c \in C} \sum_{m \in M} \sum_{\langle s,n \rangle \in t_a.acc} t_a.c.ptp_s.t_a.m \times path.c.m \times s.w \times n\end{aligned} \tag{18}$$

Finally, a task's WCET is the maximum time that needed to manage its buffers ($t_a.wcet_b$) and to execute its computations ($t_a.wcet_c$), which must not exceed its LET duration:

$$\forall t_a \in T : t_a.wcet_b + t_a.wcet_c \leq t_a.letEnd - t_a.letStart \tag{19}$$

Task schedulability. The execution time allocated to task instance t_a^i in a scheduling *slot* is tracked by the (positive) real variable $slot.t_a^i.alloc$. Enough time must be allocated for the task's WCET:

$$\forall t_a \in T \; \forall t_a^i : t_a.wcet_b + t_a.wcet_c \leq \sum_{slot \in t_a^i.slots} slot.t_a^i.alloc \tag{20}$$

For each core, the total execution time allocated to a scheduling slot ($slot.t_a^i.alloc$) must not exceed its duration ($slot.d$):

$$\forall c \in C \; \forall slot \in Slots : \sum_{t_a^i, t_a \in T} \sum_{slot \in t_a^i.slots} t_a.c \times slot.t_a^i.alloc \leq slot.d \tag{21}$$

The product $t_a.c \times slot.t_a^i.alloc$ in Equation 21 is replaced by its linearised form $t_a^i.c.slot.alloc$:

$$t_a^i.c.slot.alloc \leq t_a.c \times slot.d \tag{22}$$

$$t_a^i.c.slot.alloc \leq slot.t_a^i.alloc \tag{23}$$

$$t_a^i.c.slot.alloc \geq slot.t_a^i.alloc - (1 - t_a.c) \times slot.d \tag{24}$$

$$t_a^i.c.slot.alloc \geq 0 \tag{25}$$

8.4 Genetic Algorithm

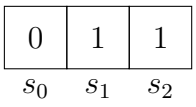
A genetic algorithm [Gol89] is a search heuristic inspired by biology to evolve a set of potential solutions toward better ones. A potential solution is called an *individual*, and a set of potential solutions is called a *population*. Each individual is comprised of at least one *chromosome* consisting of at least one *gene*. A gene encodes a system parameter, e.g., the buffering protocol selected for a signal, or a task-to-core allocation. A genetic algorithm begins by creating an initial population and then repeating the following steps:

1. Pairs of individuals are selected to have their chromosomes copied and interleaved together (*crossed over*) to produce new individuals, called *offsprings*.
2. The offsprings are mutated to increase the variability of the population. This is useful for exploring different regions of the solution space and to escape from locally optimal solutions.
3. Individuals are evaluated based on a user-defined *fitness function* and the worst individuals are removed from the population. This helps one to drive the population towards optimal solutions.

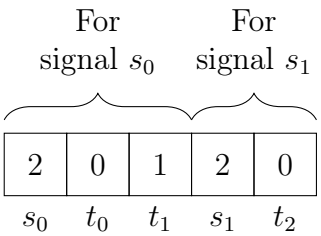
Each time these steps are repeated, a new *generation* of solutions is created. Generations are created until a user-defined *stopping criteria* is satisfied, e.g., if the improvement in solutions is insignificant, if the generation number has reached a maximum, or if the elapsed time of the genetic algorithm has reached a limit. The genes, fitness function, genetic operators, and initialisation of the population for optimising LET AUTOSAR designs are defined below.

Genes. Each chromosome has four genes, each encoded as an array, for the following system parameters:

- *Selection of a signal's buffering protocol:* an array of binary values, where each signal is mapped to an array element. A value of 0 means that PTP is selected, and 1 means that SBP is selected. The array for signal s_0 to use PTP, and for s_1 and s_2 to use SBP is thus:



- *PTP buffer-to-memory allocation:* an array of integer values from 0 to $|M| - 1$ to indicate the memory module to which a signal and a task's PTP buffer are allocated. Each signal and the tasks that use PTP are mapped to a segment of the array. The array for allocating signals s_0 and s_1 and the PTP buffers of tasks t_0 and t_1 is thus:



- *SBP buffer-to-memory allocation:* an array of integer values from 0 to $|M| - 1$ to indicate the memory module to which a signal's SBP buffer is allocated. Each signal is mapped to an array element. The array for allocating signal s_0's buffer to module 2, s_1's buffer to module 0, and s_2's buffer to module 1 is thus:

2	0	1
s_0	s_1	s_2

- *Task-to-core allocation:* an array of integer values from 0 to $|C| - 1$ to indicate the core to which a task is allocated. Each task is mapped to an array element. The array for allocating task t_0 to core 0, t_1 to core 1, and t_2 to core 2 is thus:

0	1	2
t_0	t_1	t_2

Fitness function. A fitness function returns a numerical value that evaluates how close an individual is to an optimal solution, i.e., how well Equation 4 is achieved. For an individual ψ, the fitness function $f(\psi)$ is the product of its schedulability, $\text{Sched}(\psi)$, and the amount of slack in the system, $\text{Slack}(\psi)$. A higher fitness value is better:

$$f(\psi) = \text{Sched}(\psi) \times \text{Slack}(\psi) \tag{26}$$

$\text{Sched}(\psi)$ returns a value that reflects the degree of task schedulability for ψ, i.e., the proportion of task WCETs that can be scheduled. A higher value is better. $\text{Sched}(\psi)$ calculates the total task execution time that cannot be allocated to a scheduling slot as a *deficit*. We find the minimum possible deficit with an MILP formulation, reusing the notation and definitions of Section 8.3. As in Equation 20, task WCETs are allocated to scheduling slots:

$$\forall t_a \in T \; \forall t_a^i : t_a.wcet_b + t_a.wcet_c \leq \sum_{slot \in t_a^i.slots} slot.t_a^i.alloc \tag{27}$$

The deficit of a scheduling slot on core c is tracked by the (positive) real variable $deficit.c.slot$:

$$\forall c \in C \; \forall slot \in Slots \; \forall t_a \in T \; \forall t_a^i : deficit.c.slot + slot.d - t_a.c \times slot.t_a^i.alloc \geq 0 \tag{28}$$

$$deficit.c.slot \geq 0 \tag{29}$$

The MILP objective is to minimise the total deficit:

$$\text{Minimise} : \quad deficit.total = \sum_{c \in C} \sum_{slot \in Slots} deficit.c.slot \tag{30}$$

and $deficit.total$ is used to define $\text{Sched}(\psi)$:

$$\text{Sched}(\psi) = 1 - \frac{\psi.deficit.total}{\sum_{t_a \in \psi.T} (t_a.wcet_b + t_a.wcet_c)} \tag{31}$$

Thus, we have the bound $0 \leq \text{Sched}(\psi) \leq 1$, where 0 means that no tasks in ψ can be scheduled (e.g., when all LET durations are 0), and 1 means that all tasks can be scheduled. A value in between expresses the degree to which tasks can be scheduled.

Slack(ψ) sums the amount of slack in each task's instance over one hyper-period. A positive amount is rewarded by the fitness function, while a negative amount penalises the individuals's schedulability:

$$\text{Slack}(\psi) = \sum_{t \in T} \left((t.letEnd - t.letStart) \times \frac{hp}{t.period} - \sum_{slot \in t_a^i.slots} slot.t_a^i.alloc \right) \quad (32)$$

Selection operator. When creating offsprings, two stochastic sampling methods can be used to select pairs of individuals (parents). *Tournament selection* randomly selects k individuals, and selects the fittest individual. Increasing number k increases the selection pressure. *Roulette* or *fitness proportionate selection* weighs the probability of selecting an individual on its fitness relative to others.

Crossover operator. When creating an offspring, its genes are created from the interleaving (crossover) of its parents' genes. A point is selected within the gene and all array elements from that point on are swapped between two parents. Thus, two new genes are created; one for each new offspring. For the buffering protocol and task-to-core allocation genes, the crossover point can be anywhere in the gene. For the genes encoding the PTP and SBP buffer-to-memory module allocations, the crossover point must be the one used for the buffering protocol gene. A crossover probability can be defined to control how often a crossover occurs.

Mutation operator. After an offspring has been created, its genes can be mutated by assigning random values to randomly selected array elements. For the task-to-core allocation gene, an element can be randomly assigned a value from 0 to $|C| - 1$. For the buffering protocol gene, an element can be randomly negated to switch a signal's buffering between PTP and SBP. However, the genes encoding the PTP and SBP buffer-to-memory module allocations need to be repaired. When switching a signal's buffering from SBP to PTP, the memory module of its SBP buffer is assigned to its PTP buffers. When switching from PTP to SBP, the memory module of the signal in the PTP gene is assigned to signal's SBP buffer. For the gene encoding PTP buffer-to-memory module allocations, only the array segments of the signals selected to use PTP can be mutated. For these segments, an element can be randomly assigned a value from 0 to $|M| - 1$. For the gene encoding SBP buffer-to-memory module allocations, an element can be randomly assigned a value from 0 to $|M| - 1$. A mutation probability can be defined to control how often a mutation occurs.

Initial population. A greatly simplified version of the MILP formulation, using constraint programming (CP) [FM06], is used to create an initial population of individuals. Each individual is a feasible CP solution. The simplification assumes that all cores are homogeneous, with an execution speed equal to the average of the original cores. Thus, there are only average WCET values for task execution ($t_a.wcet_{instr}$), buffer management ($wcet_{sbp}$ and $wcet_{ptp}$), and context-switching ($wcet_{cs}$). All pathways between the cores and memory modules are assumed to be constant ($path$). Task utilisation is calculated as $\frac{t_a.wcet_b + t_a.wcet_c}{t_a.period}$, which is optimistic when $t_a.period > t_a.letEnd - t_a.letStart$. Lastly, the overhead for managing PTP or SBP buffers for a signal is assumed to be a constant.

The CP formulation uses the same MILP constraints to model SBP and PTP buffering (Equations 5–11), task-to-core allocations (Equation 12), and to check that a task's WCET does not exceed its LET duration (Equation 19). The worst-case time for task t_a to manage

its buffers is stored in the (positive) real constant $t_a.wcet_b$. It is the sum of the maximum buffer management overheads due to SBP or PTP:

$$\forall t_a \in T : t_a.wcet_b = 2 \times wcet_{cs} + \sum_{\langle s,n \rangle \in t_a.acc} \max\left(wcet_{sbp}, wcet_{ptp} + 2 \times path \times s.w\right) \quad (33)$$

The worst-case time for task t_a to execute its computations is stored in the (positive) real constant $t_a.wcet_c$, which is the sum of the average worst-case time to execute t_a's instructions ($t_a.wcet_{instr}$), to context-switch to and from t_a ($wcet_{cs}$), and for t_a to access its buffered signals.

$$\forall t_a \in T : t_a.wcet_c = t_a.wcet_{instr} + wcet_{cs} \times t_a.cs + \sum_{\langle s,n \rangle \in t_a.acc} path \times s.w \times n \quad (34)$$

A core is completely utilised (value equal to 1) if it needs to spend all its time executing tasks. The utilisation of a core must not exceed 1:

$$\forall c \in C : 1 \geq \sum_{t \in T} t.c \times \frac{t.wcet_b + t.wcet_c}{t.period} \quad (35)$$

8.5 Scheduling Hints and Reducing End-to-End Response Times

After MILP or the genetic algorithm has found a solution, a physical task schedule can be constructed for each core based on the task-to-core allocations. The LET start and end routines needed for PTP (see Section 8.1) must be included in the task schedule. Although the technical details for constructing physical task schedules are beyond the scope of this report, we suggest some scheduling hints that facilitate further memory optimisations for SBP. We observe that additional buffer elements are required when writers and readers are overlapped with each other and need different snapshots of a signal (see Algorithm 2, line 35). If the writers and readers can be scheduled without overlapping, then less buffer elements may be needed for SBP. For example, if a writer instance t_w^i is overlapped with a reader instance t_r^j, we know that t_r^j must use the output from a prior instance of t_w^i, e.g., t_w^{i-1}. Thus, if t_r^j can be scheduled to execute completely before t_w^i, then t_r^j's buffer element could be reused by t_w^i. Of course, t_r^j should not be scheduled too early such that it overlaps with t_w^{i-1}. Overall, the strategy is to schedule readers as early as possible, and writers as late as possible. Unfortunately, tasks that write and read to signals can only be optimised for writing or reading, and not both.

Algorithm 5 presents a heuristic that decides whether it is better to schedule a task for writing or for reading. Let $t_a.writes \subseteq S$ and $t_a.reads \subseteq S$ contain the set of signals that task t_a writes and reads, respectively. The heuristic, called SchedulingHints, begins by recording the tasks that do not read any signals as being better scheduled for writing (line 3). The tasks that do not write to any signals, or tasks that write to and read from the same signal, are better scheduled for reading (line 4). Moreover, tasks that read signals written by tasks already in *SchedAsWriters* can only be scheduled for reading (line 5). For the remaining tasks (line 6), which read and write to different signals, we are only interested in those with the least number of signal writes (lines 7), and that do not have successors already in *SchedAsWriters* (lines 8). This is because we choose to schedule these tasks for writing (lines 9) and their successors for reading (lines 10). Note that the algorithm could terminate without recording a scheduling preference for some tasks. In such cases, these tasks can be scheduled for writing or for reading.

As mentioned at the end of Section 2.5, the system's overall end-to-end response times can be shortened by scaling down the timing parameters of all tasks until a task no longer

Algorithm 5 SchedulingHints returns the tasks that are better scheduled for writing and the tasks that are better scheduled for reading.

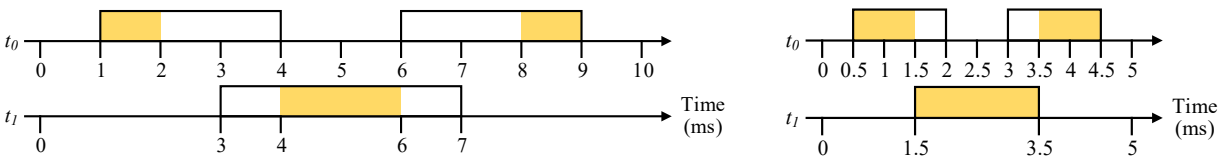
Input: T (all tasks)
Output: $SchedAsWriters$ (tasks better scheduled for writing), and $SchedAsReaders$ (tasks better scheduled for reading)

// Initially, there are no scheduling hints.
1: $SchedAsWriters \leftarrow \emptyset$
2: $SchedAsReaders \leftarrow \emptyset$

// Tasks that do not read any signals are better scheduled for writing.
3: $SchedAsWriters \leftarrow \{t_a \in T \mid t_a.reads = \emptyset\}$

// Tasks that do not write to any signals, or tasks that write to and read from the same
// signal, are better optimised for reading.
// The successors of the tasks in $SchedAsWriters$ are also better scheduled for reading.
4: $SchedAsReaders \leftarrow \{t_a \in T \mid t_a.writes = \emptyset \text{or} t_a.writes \cap t_a.reads \neq \emptyset\}$
5: $\qquad\qquad \cup \{t_c \in \mathrm{R}(s) \mid s \in t_b.writes, t_b \in SchedAsWriters\}$

// Decide which of the remaining tasks are better scheduled for writing or for reading.
// Only consider the tasks with successors that are not scheduled for writing.
6: **while** $\exists t_a \in T \setminus (SchedAsWriters \cup SchedAsReaders)$
7: $\quad$ **where** $|t_a.writes|$ **is** minimal
8: $\quad$ **and** $\{t_b \in \mathrm{R}(s) \mid s \in t_a.writes\} \cap SchedAsWriters = \emptyset$
do
9: $\quad SchedAsWriters \leftarrow SchedAsWriters \cup \{t_a\}$
10: $\quad SchedAsReaders \leftarrow SchedAsReaders \cup \{t_b \in \mathrm{R}(s) \mid s \in t_a.writes\}$
11: **end while**



(a) Unscaled physical task schedule. All task instances have 2 ms of slack.

(b) Scaled down physical task schedule. Task instance t_1^0 no longer has any slack.

Figure 14: Example of scaling down a physical task schedule to shorten a system's overall end-to-end response times. The tasks are allocated to the same core, and two instances of t_0 and one instance of t_1 execute during the 10 ms hyper-period.

Table 10: Timing parameters (in ms) of two LET tasks allocated to the same core.

Task	Period	LET Start	LET End	WCET
t_0	5	1	4	1
t_1	10	3	7	2

has any slack. Figure 14a presents a small physical task schedule for tasks t_0 and t_1 on the same core. Both tasks use SBP and their original and scaled task timing parameters are given in Table 10. In the physical task schedule, each task instance has a slack of 2 ms. The maximum scaling of the timing parameters is determined by the task instance with the least amount of slack, relative to its LET duration:

$$MaxScaling = \min\left(\left\{\frac{t^i\text{'s slack}}{t.letEnd - t.letStart} \;\middle|\; \forall t^i, t \in T\right\}\right) \tag{36}$$

For Figure 14a, $MaxScaling = \min\left(\{\frac{2}{3}, \frac{2}{3}, \frac{2}{4}\}\right) = 0.5$. Thus, the physical task schedule can be scaled down by 50% as shown in Figure 14b. This approach to improving end-to-end response times is applicable to LET tasks that use PTP by only scaling the timing parameters of the computation task.

8.6 Refining the SBP Buffering Schedules

Once the physical task schedule has been constructed, better estimates of SBP buffer lifetimes can be made. This is because the start and end times of a task's execution are likely shorter than its LET. Less buffer elements may be needed if the execution of the writers and readers do not overlap and if they execute sequentially with respect to the data-flow. Thus, SbpBufferingSchedules of Algorithm 4 is modified to consider the allocated execution time of the task instances. Let $t_a.execStart$ and $t_a.execEnd$ denote the start and end of t_a's allocated execution time, respectively. On line 3, $t_r^i.letEnd$ is replaced by $t_r^i.execEnd$ because readers no longer need their buffer elements after they have completed their computations. On line 5, $t_w^i.letStart$ is replaced by $t_w^i.execStart$ because writers only need their buffer elements when they start their computations. Line 22 in Algorithm 3 is updated identically. For the signals that use SBP, SbpBufferingSchedules is then run once more to generate possibly more memory efficient buffering schedules, and more pairs of signals with disjoint buffer lifetimes.

8.7 Merging the SBP Buffer Memories

The amount of memory needed for SBP can be reduced further by allowing signals with buffers that have disjoint lifetimes (over the hyper-period) to share the same memory, i.e., to merge their buffers. For each memory module, a graph of its allocated signals with disjoint buffer lifetimes is created and analysed. Let $G = \langle S, E\rangle$ be such a graph, where $s \in S$ is a signal, and $\langle s_0, s_1\rangle \in E$ is an undirected edge that connects two signals that have disjoint buffer lifetimes. A *clique*, $K \subseteq G$, which is a fully connected subgraph, represents signal buffers that can be merged together. Multiple cliques may be found and some may overlap with others. When a clique is merged, it will prevent all partially overlapping cliques from merging. Thus, the selection and the order in which cliques are merged has a significant impact on the possible amount of memory reduction.

Algorithm 6 describes the selection of signal buffers for merging. For each memory module (line 1), its allocated signals are retrieved from the MILP or genetic algorithm solution (line 2), and a graph of disjoint buffer lifetimes is created (line 3). While the graph still contains cliques (line 6), a clique is selected and its signals are recorded for merging (line 7), and the clique is removed from the graph (line 8).

When finding a clique of signal buffers to merge, we wish to weigh them by their potential memory reduction. When buffers are merged, the resulting buffer size is equal to the maximum of the individual buffers. Thus, the memory reduction for a clique K can be

calculated in relative ($savR_K$) or absolute ($savA_K$) terms:

$$savR_K = 1 - \frac{\max\left(\{s.size \times s.n \mid s \in K\}\right)}{\sum_{s \in K}(s.size \times s.n)} \tag{37}$$

$$savA_K = \sum_{s \in K}(s.size \times s.n) - \max\left(\{s.size \times s.n \mid s \in K\}\right) \tag{38}$$

When choosing buffers to merge, a balance between high relative savings and high absolute savings is preferred. For example, a 75% saving of 32 bytes is not necessarily better than a 50% saving of 128 bytes. Hence, the relative and absolute savings can be weighed together, e.g., $savW_K = savR_K \times savA_K$, where a greater $savW_K$ indicates better potential for memory reduction. Algorithm 7 presents a heuristic, called GetClique, for finding the greatest weighted clique in a graph. Line 3 relies on existing algorithms [BK73] to find all cliques in a graph, which is known to be NP-hard [Kar72]. Lines 4–12 searches for the clique with the greatest weight.

Algorithm 6 MergeSbp returns groups of signal buffers that can be merged.

Input: *AllDisjBuffs* (pairs of signals with disjoint buffer lifetimes), S (all signals), and *Solution* (optimisation solution from MILP or genetic algorithm)

Output: *AllMergedBuffs* (groups of signals that can merge their buffers)

1: **for** $m \in Solution.M$ **do**
   // Construct graph of disjoint buffer lifetimes for memory module m.
2: $S_{sbp} \leftarrow \{s \in S \mid Solution.sbp_s = 1 \wedge Solution.sbp_s.m = 1\}$
3: $G \leftarrow \langle S_{sbp}, \{disjBuffs \in AllDisjBuffs \mid disjBuffs \subseteq S_{sbp}\}\rangle$

4: $MergedBuffs_m \leftarrow \emptyset$
5: $K \leftarrow \text{GetClique}(G)$
6: **while** $K \neq \emptyset$ **do**
      // Merge all signal buffers in K.
7: $MergedBuffs_m \leftarrow MergedBuffs_m \cup \{K\}$

      // Remove clique K from the graph.
8: $G \leftarrow \langle G.S \setminus K, \{disjBuffs \in G.E \mid disjBuffs \cap K = \emptyset\}\rangle$
9: $K \leftarrow \text{GetClique}(G)$
10: **end while**

11: $AllMergedBuffs \leftarrow AllMergedBuffs \cup \{MergedBuffs_m\}$
12: **end for**

8.8 Discussion

After all design and deployment phase optimisations have been applied to a LET-based AUTOSAR design (see Section 6 for the overview), we have the physical task schedule of each core and the SBP buffering schedules of the relevant signals. From these, the final implementation can be generated: (1) the PTP buffers are created, (2) the signals chosen to use SBP are converted into SBP buffers, (3) the LET routines and variables needed for PTP and SBP buffer management are created, (4) the tasks are modified to

Algorithm 7 GetClique returns a clique of signals with the greatest weighted memory reduction.

Input: G (graph of signals with disjoint SBP buffer lifetimes)
Output: K^{found} (clique with the greatest weight)

$savW^{found} \leftarrow 0$
$K^{found} \leftarrow \emptyset$
$K_{All} \leftarrow \text{GetAllCliques}(G)$

for $K \in K_{All}$ **do**
    $savR_K \leftarrow 1 - \dfrac{\max\left(\{s.size \times s.n \mid s \in K\}\right)}{\sum\limits_{s \in K} s.size \times s.n}$
    $savA_K \leftarrow \sum\limits_{s \in K} (s.size \times s.n) - \max\left(\{s.size \times s.n \mid s \in K\}\right)$
    $savW_K = savR_K \times savA_K$

    if $savW^{found} < savW_K$ **then** // Clique with greater weight found.
        $savW^{found} \leftarrow savW_K$
        $K^{found} \leftarrow K$
    end if
end for

access the buffered signals, and (5) the physical task schedule of each core is transformed into an AUTOSAR schedule table. Any reduction of a system's end-to-end response times necessarily involves modifying the timing behaviour of the original AUTOSAR design. Thus, complete preservation of LET communication semantics is not possible, but the advantage of our scaling approach is that data-flow is preserved.

The deployment optimisations can be refined further by analysing AUTOSAR designs at a finer level of detail. For example, disjoint buffer lifetime analysis is performed over the entire hyper-period, but it could be improved by identifying intervals within the hyper-period where buffer lifetimes are disjoint and can therefore be merged. Task allocation could also be extended to task instances, allowing communicating task instances to execute closer together. However, by analysing at a finer granularity, the search space is increased considerably.

To reduce the exploration space of the deployment optimisations, the merging of SBP buffers is performed after MILP or the genetic algorithm has selected the buffering protocol of each signal and the signal buffer-to-memory module allocations. If the merging of SBP buffers is also considered by MILP or the genetic algorithm, then more solutions may be found for memory-constrained AUTOSAR designs. The heuristics for suggesting the scheduling of tasks as writers or readers, and for the merging SBP buffer memories can be refined based on experimental data.

9 Tooling

The prototyping and evaluation of a selection of the proposed algorithms and heuristics (see Sections 7 and 8) is carried out using the TA Tool Suite [Vec18] (version 17.4). The TA Tool Suite uses a simulation-based approach to assist AUTOSAR designers in modelling, designing, and analysing the timing behaviour of event-triggered or time-triggered multi-core automotive software. The execution of an AUTOSAR design can be simulated for a user-specified duration, and the resulting event trace can be viewed as an interactive Gantt chart. The event trace can be processed to return a variety of execution metrics, e.g., core execution times, memory access times, task execution times, and deadline violations. The TA Tool Suite has been extended to support the PTP and SBP buffering protocols, the modification of LET tasks to access buffered signals, the inclusion of LET start and end routines in the schedule tables, the simulation of PTP or SBP buffering, and the evaluation of buffering metrics from the simulation traces.

This section describes the abstract software and hardware model that the TA Tool Suite supports, the algorithms and heuristics from Sections 7 and 8 that are prototyped, and the buffering-specific evaluation metrics that are used for the synthetic benchmarking (see Section 10) and the FMTV case study (see Section 11).

9.1 Software and Hardware Model

The TA Tool Suite's system model allows the specification of runnables and tasks, task schedulers, and operating systems. A runnable is defined as a sequence of abstract instruction blocks, interleaved with signal accesses. The execution time of an instruction block or signal access is resolved at simulation time, because it depends on the speed of the processor cores, memory modules, and latency of the interconnects (pathways). Moreover, the number of instructions to execute can be defined as a probability distribution, e.g., the Gauss or Weibull distributions, and be based on observed execution times from real software. Each signal has a data type and size, and is allocated to a memory module. A data age constraint can be defined for any combination of signals and runnables. Runnables are allocated to tasks and a runnable execution order can be defined for each task. Event-chains can be specified to investigate end-to-end response times.

An operating system can be selected to manage one or more cores, and overheads can be set for various types of context-switches or interrupt service routines. An operating system can have one or more task schedulers, each with its allocation of tasks. Each scheduler is allocated to a core. Scheduling algorithms, e.g., fixed-priority or rate-monotonic, can be selected for each task scheduler, but we are only interested in the hyper-period scheduling algorithm. The system model supports many other software aspects that are not necessary for this work, e.g., (1) the use of mutexes, semaphores, or spin-locks as data protection mechanisms, (2) the runnable execution call tree that allows runnables to be conditionally executed for a more precise modelling of execution behaviour, or (3) the periodic or sporadic triggering of tasks with stimuli or operating system events.

Processors, memory modules, and their pathways are specified in the system model. A processor has one or more cores, each with its own execution speed. A memory module has a fixed capacity, data-width, write and read latencies, and an operating frequency. The memory modules are connected to the cores through interconnects and their latencies depend on the selected arbitration policy, e.g., fixed-priority or round-robin. For our work, we assume a multi-core hardware architecture like the one depicted in Figure 1d, and that signal accesses are not cached.

The system model has been extended with the ability to define LET tasks, and a physical

task schedule for each scheduler. For LET tasks, the initial offset, activation offset, LET duration, and period can be specified (see Figure 3). A plug-in has been implemented to automatically modify LET tasks into ordinary tasks by replacing all signal accesses with buffered equivalents, by inserting special runnables to model the overhead of updating SBP buffer indexes, by creating special tasks to model the LET start and end routines for PTP buffering, and by creating the signal buffers. Another plug-in was implemented to automatically calculate the hyper-period, to generate physical task schedules from a software model, and to insert the schedules into the software model. Each task schedule is stored as a list of timestamps with a scheduling action, e.g., at 0 *ms* release task t_0. At the end of each hyper-period, a fixed *preparation time* is specified for the reinitialisation of SBP buffers so that their initial buffer element holds the correct value. Thus, the periodicity of a task schedule is slightly longer than its hyper-period.

9.2 Prototyped Optimisations

From the overview of the proposed buffering optimisations described in Section 6, Steps 1 and 6, and partially Steps 3 and 5 have been prototyped. Step 1 relates to the design phase optimisations described in Section 7, where *SbpBufferingSchedules* (see Section 7.3) constructs an SBP buffering schedule for each signal and finds buffers with disjoint lifetimes. *GetWriterUses* (see Section 7.2) has been implemented to suppress unnecessary signal writes. For Step 3, only the construction of unoptimised physical task schedules is supported by the TA Tool Suite. Step 5 relates to the merging of SBP buffers with *MergeSbp* (see Section 8.7). However, a simplified version has been implemented, where pairs of SBP signal buffers are merged if they have disjoint lifetimes and the same buffer sizes. This has been translated into a graph colouring problem and a greedy algorithm was implemented. For Step 6, a plug-in has been implemented to insert the necessary buffering-related code, signal buffers, and physical task schedules into the system model.

Because only some of the proposed algorithms and heuristics have been implemented, further assumptions on the system model are needed:

- A model uses either PTP or SBP for all its signals. It is not possible to use a mix of PTP or SBP in the same model.
- Each SBP buffer is allocated to the same memory module as its original signal.
- A task's PTP buffer is allocated to the local memory of the task's allocated core.
- A task's SBP buffer indexes are allocated to its stack, and the indexes are updated in each task instance.
- Because signal buffer-to-memory module allocations are fixed, the TA Tool Suite only warns the user when a memory module's capacity is exceeded by its allocated buffers.

9.3 Evaluation Metrics

The following metrics are used to evaluate the impact that the PTP and SBP buffering protocols and the implemented SBP buffering optimisations have on LET-based multi-core AUTOSAR designs.

Total signal and buffer element count: The total number of signals and buffer elements in the system. This metric indicates the amount of complexity that the chosen buffering protocol has added to the system. The added complexity could increase

the implementation effort needed to generate a deployment. This metric is calculated after the final system model has been generated.

Total signal and buffer memory: The total amount of memory needed for all signals and buffers. This metric indicates the amount of memory that is needed to preserve the LET communication semantics, and is calculated after the final system model has been generated.

Total buffer management time: The sum of the time needed to manage all buffers during a simulation run. For PTP buffer management, it is the execution time of the LET start and end routines. For SBP buffer management, it is the time to update the buffer indexes and to reinitialise the buffers after each hyper-period, and when the local programming style is used, the time to read a signal into a task's local variable. Arbitration delays are considered in every memory access. This metric is calculated after simulation.

Total signal access time: The sum of the times needed during a simulation run to access the buffered signals in the task computations, to access signals and buffers in the LET start and end routines for PTP buffering, and to access variables when updating buffer indexes for SBP buffering. Arbitration delays are considered in every memory access. This metric is calculated after simulation.

Processor utilisation: The utilisation of a core is the percentage of time that it spends executing tasks, scheduler, accessing signals, and accessing buffers. The processor utilisation is the average utilisation of all its cores. This metric is calculated after simulation.

Table 11: Task periods for the airbag, chassis, and engine management systems, with hyper-periods of $1,000\ ms$, $10\ ms$, and $1,000\ ms$, respectively

	Occurrence (%)		
Period (ms)	**Airbag**	**Chassis**	**Engine**
0.5	5	·	·
1	33	18	10
2	·	3	16
2.5	·	3	·
5	10	41	40
10	21	35	10
20	5	·	3
40	3	·	·
50	·	·	4
100	10	·	·
200	·	·	10
400	3	·	·
1,000	10	·	7

10 Synthetic Benchmarking

The capabilities of the implemented buffering optimisations (see Section 9.2) are explored with synthetic benchmarks that are representative of AUTOSAR designs from the automotive industry. The evaluation metrics of unbuffered, PTP buffered, and SBP buffered AUTOSAR systems are compared. Eight different AUTOSAR designs for managing an airbag, chassis, and engine have been collected and analysed. Based on industrial experiences with working on similar designs, software parameters characterising the three categories of AUTOSAR designs have been derived and are listed in Tables 11 and 12. Table 11 defines the task periods observed in each category and their occurrence as a percentage of the total number of tasks. Sporadic tasks, e.g., interrupt service routines, or tasks triggered by aperiodic events, are ignored. Table 12 defines the range of signals, runnables, and tasks observed in the AUTOSAR designs. The processor utilisations (which includes the execution of sporadic tasks) have been derived by simulating each design in the TA Tool Suite.

10.1 Benchmarking Workflow

An existing model generator tool for the AMALTHEA tool platform [Inf17] has been extended to generate LET-based AUTOSAR designs for the TA Tool Suite. Figure 15 shows a portion of the configuration interface, where the randomness of each parameter is defined by a probability distribution. Twenty random models of each system category have been generated from the parameters in Tables 11 and 12. Every task has a random LET duration equal to $50--100\%$ of their period, and an activation offset and initial offset equal to zero. All signals are accessed directly by the runnables without any data protection mechanisms. For each model, data age constraints are randomly added to twenty percent of the signals. The data age duration has a uniform value between 3–7 times the period of the writer task.

All the generated models use an identical hardware platform, similar to Figure 1d. It has a single processor with three homogeneous cores at a fixed execution speed of 300 MHz.

Table 12: Characteristic software parameters

Parameter	Airbag	Chassis	EMS
Signal data size	6–32 bits	6–32 bits	6–32 bits
Number of signals	2,000–4,500	1,000–2,000	2,000–5,000
Number of signal accesses	2,373–9,853	11,346–23,215	1,615–11,868
Number of runnables	491–1,858	2,124–4,278	342–2,219
Instructions per runnable	1,000–100,000	100–1,000	1,000–50,000
Number of tasks	20–40	25–30	25–40
Processor utilisation	40–80%	40–85%	40–65%

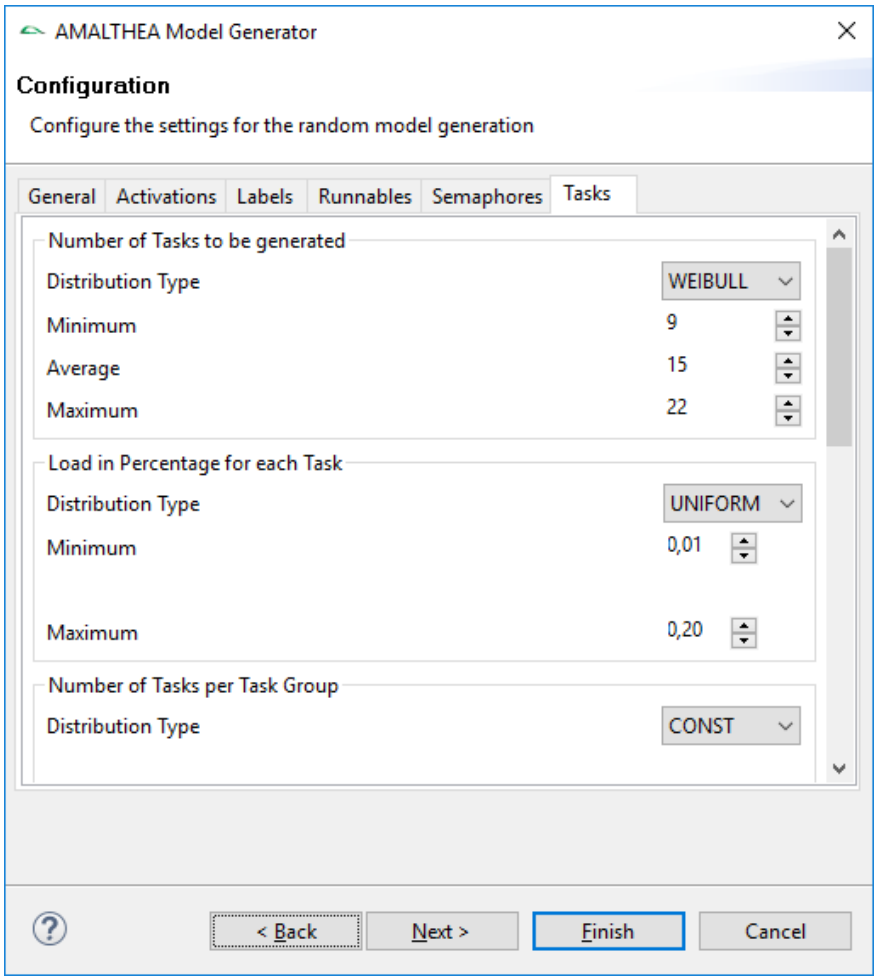


Figure 15: Screenshot of the model generator configuration interface.

Each core has access to its own local memory module via a fast local bus (5 ns access time), and access to a shared global memory module via a slower bus (10 ns access time). First-come, first-serve arbitration is used for the bus requests. All memory modules have a data width of 64 bits.

After the unbuffered models have been synthesised, PTP and SBP versions are generated with the assumptions listed in Section 9.2. The PTP buffering protocol, without any PTP-specific optimisations, is applied to the unbuffered models to create *PTP buffered models*. The SBP buffering protocol and the implemented buffering optimisations are applied to the unbuffered models to create *SBP buffered models*. Because the suppression of unnecessary writes only apply to signals with the local programming style, but the unbuffered models use the global programming style, we add local variables (allocated to task stacks) to model the cost of using the local programming style. More precisely, for each signal to which a task writes, a local variable is added. Thus, two versions of the SBP buffered models are generated: *SBP global* where all signals use the global programming style, and *SBP local* where all signals use the local programming style. We do not create models where a mix of PTP and SBP buffering protocols are applied.

10.2 Preliminary Results

This section uses the evaluation metrics from Section 9.3 to compare the unbuffered and buffered synthetic models. The simulated run-time for each model is three times its hyper-period, i.e., $3 \times hp$.

Total signal and buffer element count. Figures 16–18 show that the SBP buffered models require less signal and buffer elements than the PTP buffered models. The SBP buffered models benefit from the proposed buffer memory optimisations (static buffering protocol and buffer merging), resulting in less buffer elements needed when compared with PTP buffering. However, SBP buffering can require a significant number of buffer indexes, which are drawn on top of the SBP buffering results in Figures 16–18. The number of buffer indexes varies with each model, because it depends on the number of tasks and the variety of signals that each task accesses. Compared to the SBP global buffered models, the SBP local buffered models require additional local variables. Overall, the SBP buffered models are more complex to implement than the PTP buffered models.

Total signal and buffer memory. Taking signal data sizes into account, Figures 19–21 show the total memory that is needed for the signals and buffers. Significantly more memory is needed for the PTP buffered models when compared to the SBP global buffered models. This is again due to the SBP buffering optimisations and that PTP buffering always requires a buffer for each writer and reader of a signal. A greater difference is observed when larger signal data sizes are used. Even when buffer indexes of 8 bits in size are taken into account, SBP buffering usually needs less memory than PTP buffering. The actual memory needed at run-time is less because buffer indexes are allocated to the task stacks, so their memory is freed when tasks terminate. The same reasoning applies to the local variables in the SBP local buffering models.

Processor utilisation. Figure 22 shows that the average processor utilisation between the PTP and SBP buffered models of the airbag and engine management systems are nearly the same. Thus, there is no performance advantage between using PTP or SBP buffering for these systems. A noticeable difference between PTP and SBP buffering can be seen for the

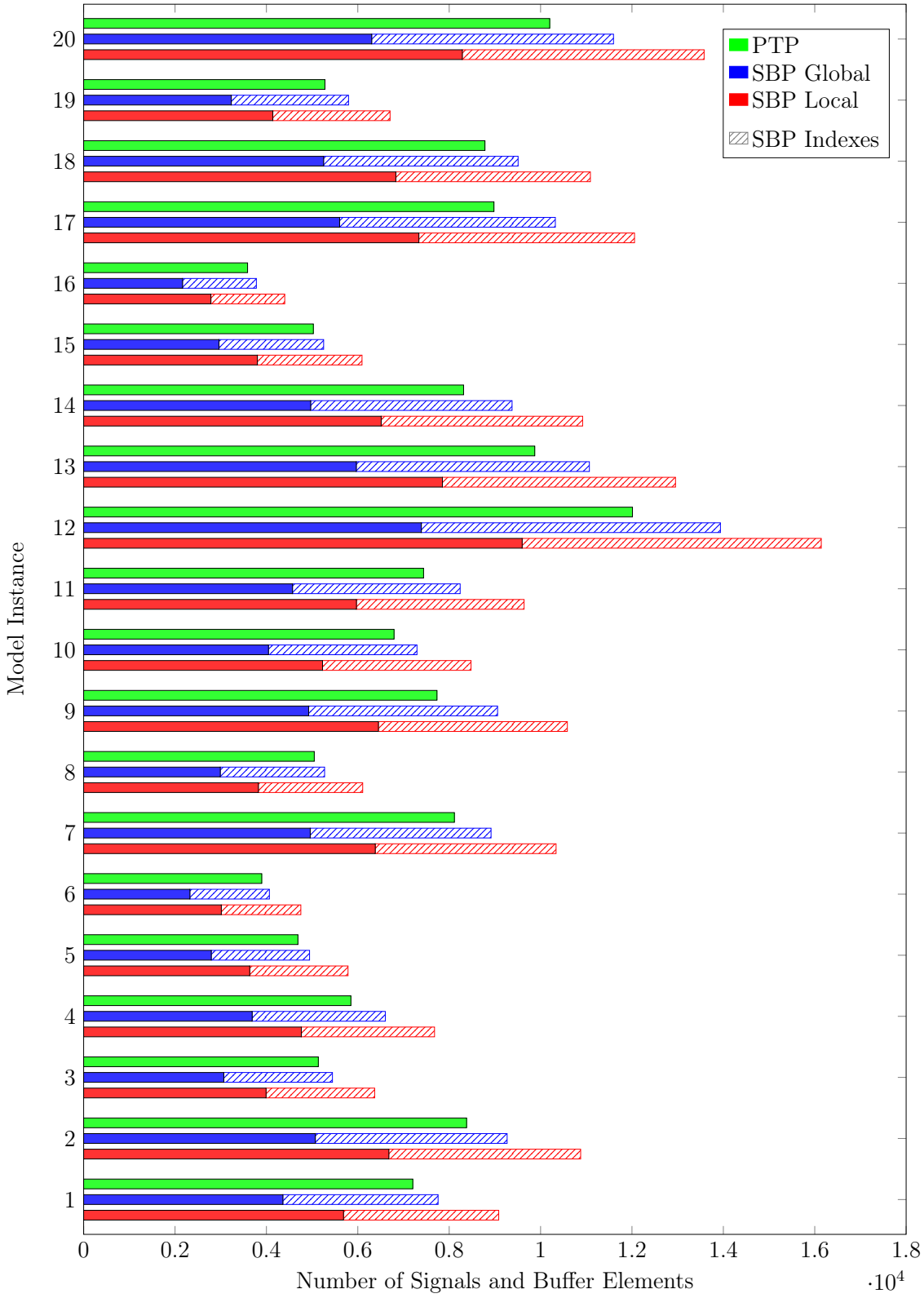


Figure 16: Total signal and buffer element counts for the airbag models.

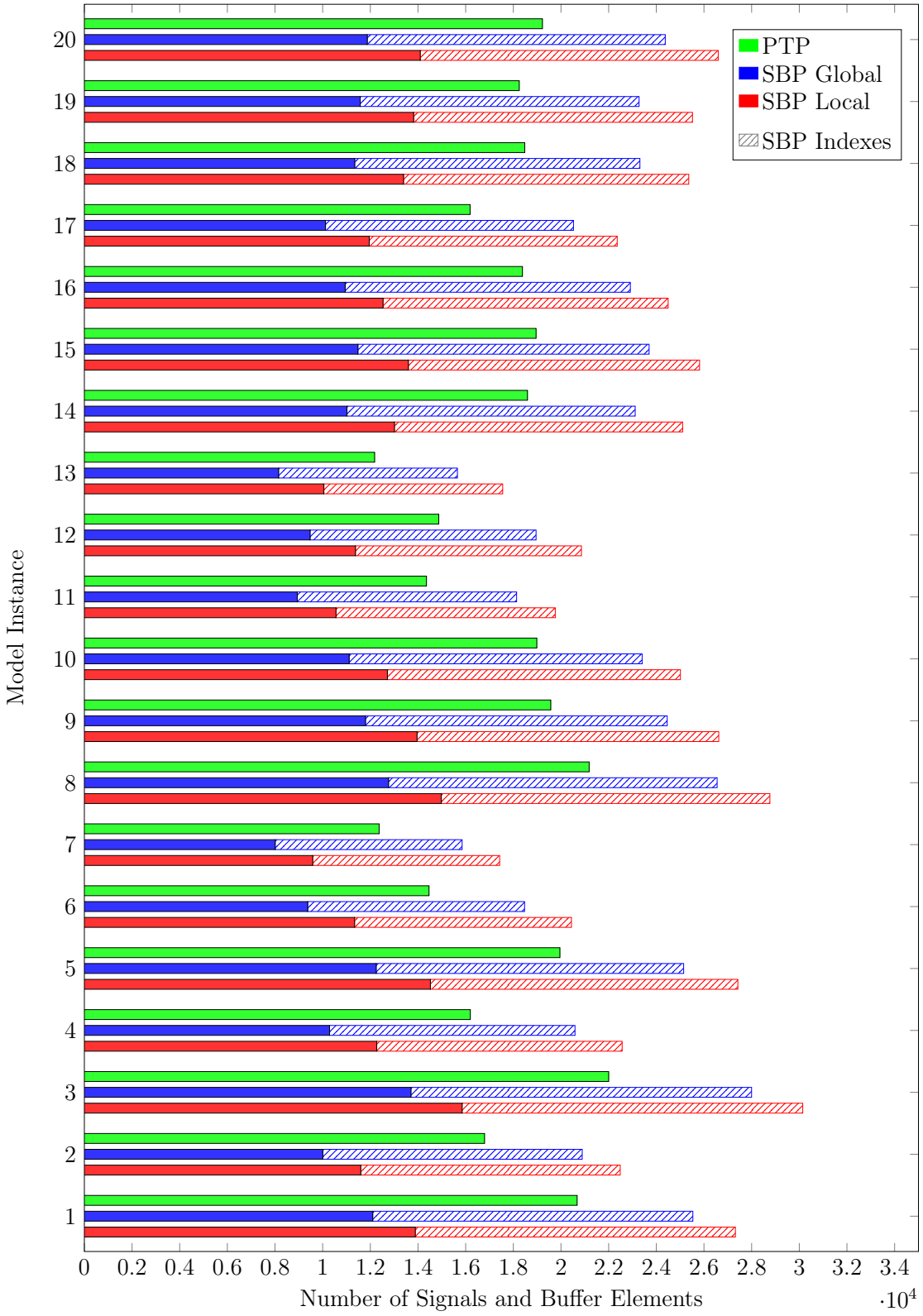


Figure 17: Total signal and buffer element counts for the chassis models.

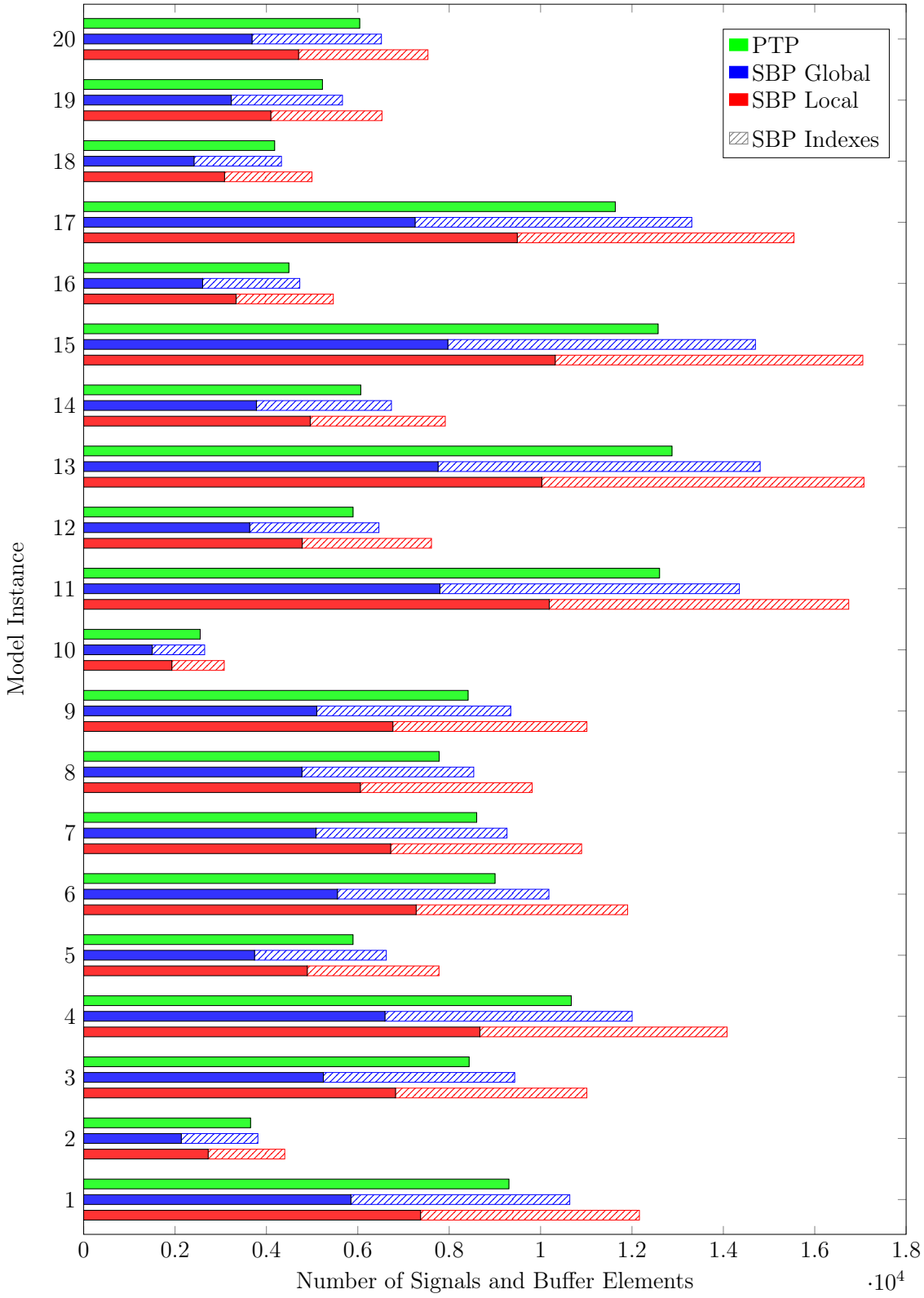


Figure 18: Total signal and buffer element counts for the engine models.

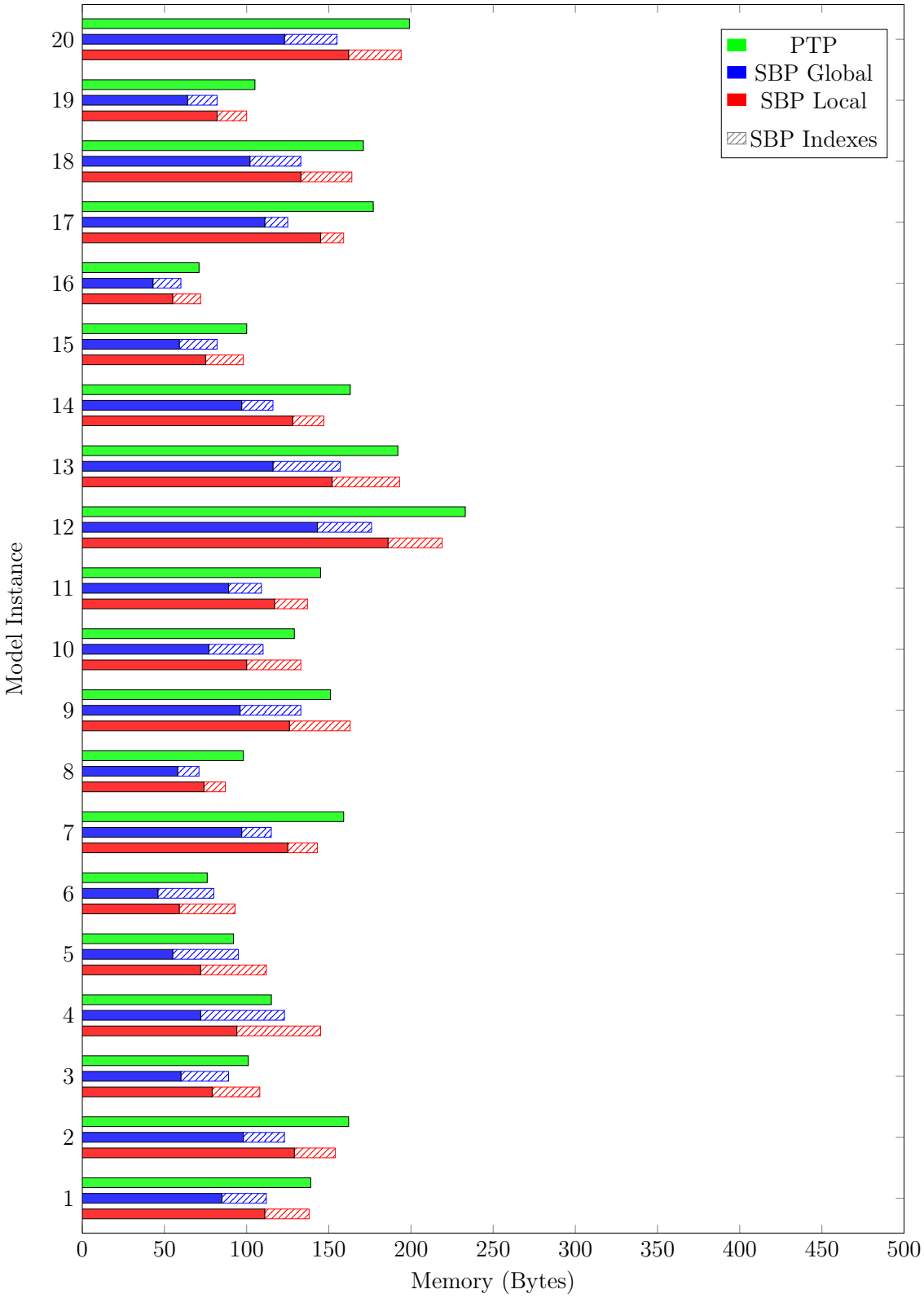


Figure 19: Total memory needed for the signals and buffers in the airbag models.

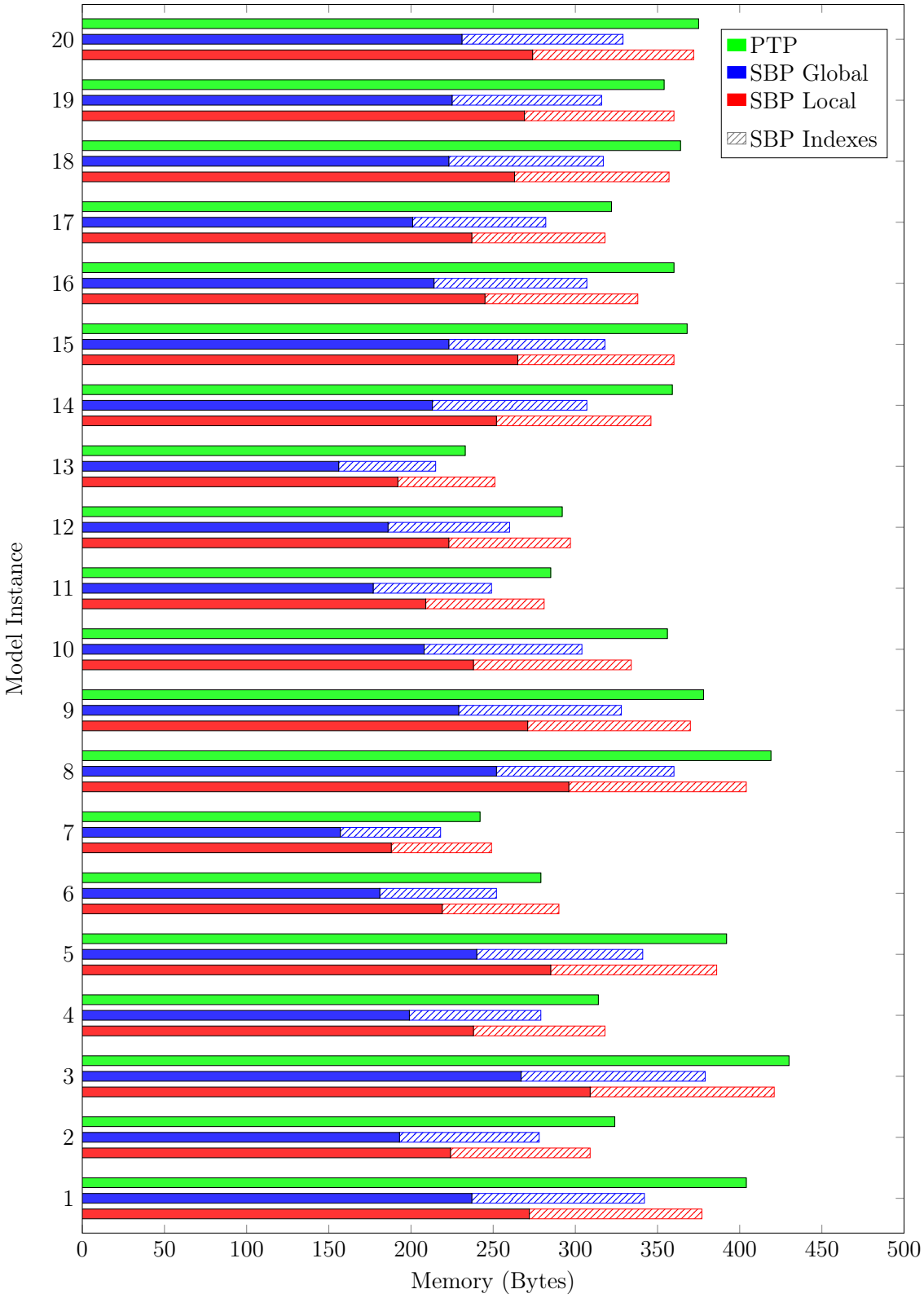


Figure 20: Total memory needed for the signals and buffers in the chassis models.

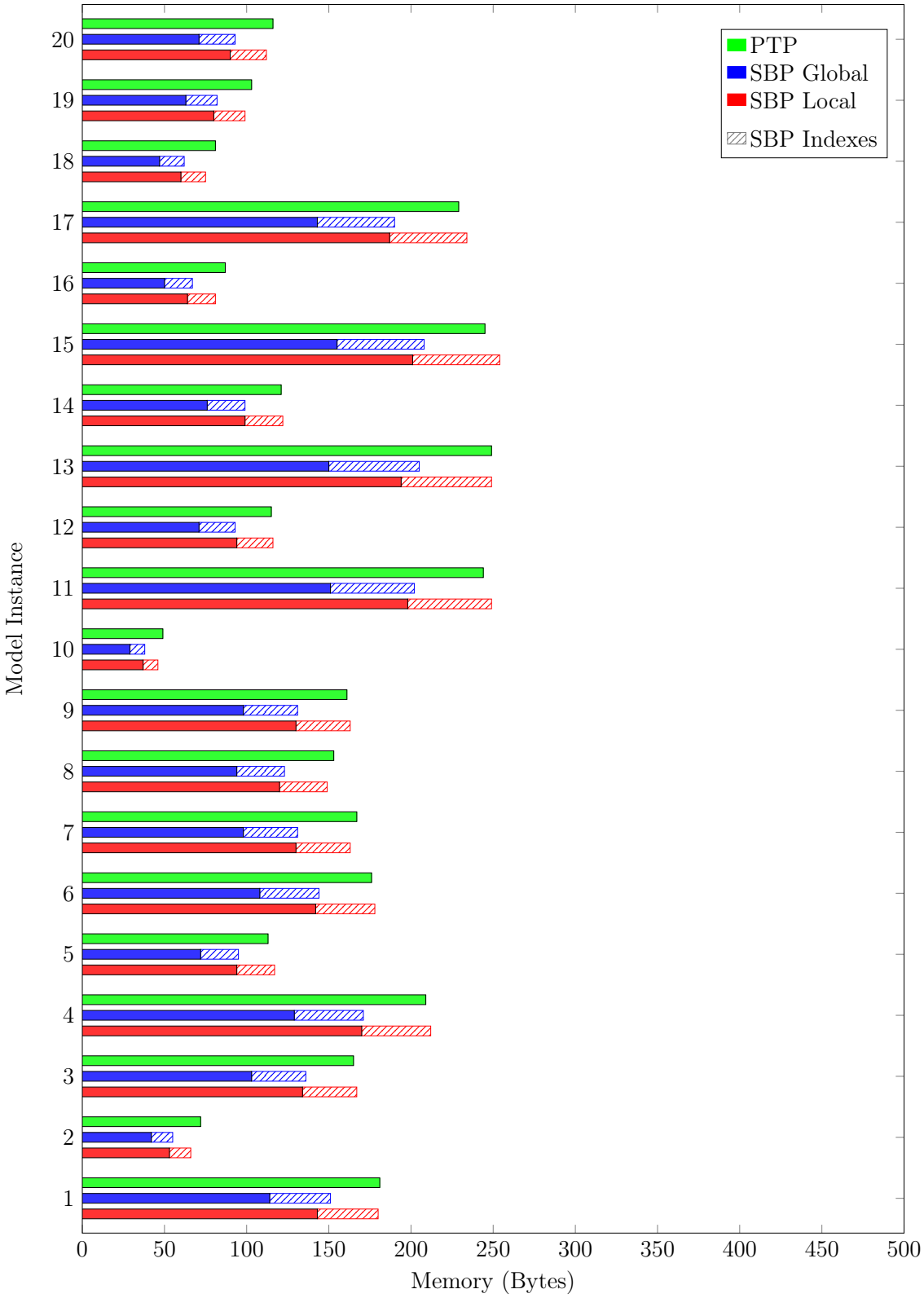


Figure 21: Total memory needed for the signals and buffers in the engine models.

chassis system. The higher processor load of the PTP buffered models suggests that PTP buffering incurs a higher overhead than SBP buffering.

Total signal access time. Figure 23 shows that more time is needed to complete all signal accesses with PTP buffering than with SBP buffering, although the time difference is relatively small. This appears to be counter-intuitive because accessing a local memory module is normally faster than accessing a global memory module. However, the reported total signal access time includes the time to access the signals and buffers in the LET start and end routines for PTP buffering, and the time to access variables when updating buffer indexes for SBP.

Total buffer management time. Figure 24 shows that PTP buffering requires at least twice the time of SBP global buffering to complete all their signal accesses. The overhead for SBP local buffering, however, is slightly better than that of PTP buffering. First, the alignment of multiple LET start and end routines across the cores could cause high bus contention, resulting in higher arbitration delays for the PTP buffering model. Second, the SBP local buffering models have to perform additional writes from the local outputs to the signals.

10.3 Discussion

The preliminary evaluation presented in this section demonstrates that the design phase optimisations alone make SBP buffering a competitive alternative to PTP buffering, which is typically used for LET-based AUTOSAR systems. The results showed that SBP buffering uses less memory and has shorter signal access times than PTP buffering. Greater differences in the evaluation metrics between the PTP and SBP buffered models could have been elicited by increasing the number of signal accesses within each runnable (and task). SBP global buffering is better than SBP local buffering in all evaluation metrics when the suppression of unnecessary writers and the merging of signal buffers yields insignificant savings.

SBP buffering relies on the use of buffer indexes to ensure that task instances access the correct buffer elements, which increases the stack space needed for each task. This is mitigated by the fact that only stack space is needed for the indexes during task execution, which is released on task termination. The total memory needed for SBP buffer indexes can be reduced by using indexes that are smaller than 8 bits. Devising a physical task scheduler with a shorter hyper-period will help to reduce the number of task instances that have to be counted.

From an implementation point of view, SBP buffering requires buffering schedules to be regenerated whenever a task is added or removed from the design. In addition, the reinitialisation of SBP buffers at the end of each hyper-period need to be updated. PTP buffering only requires the LET start and end routines of the affected task to be updated, leading to better maintainability and perhaps less implementation effort in the long term.

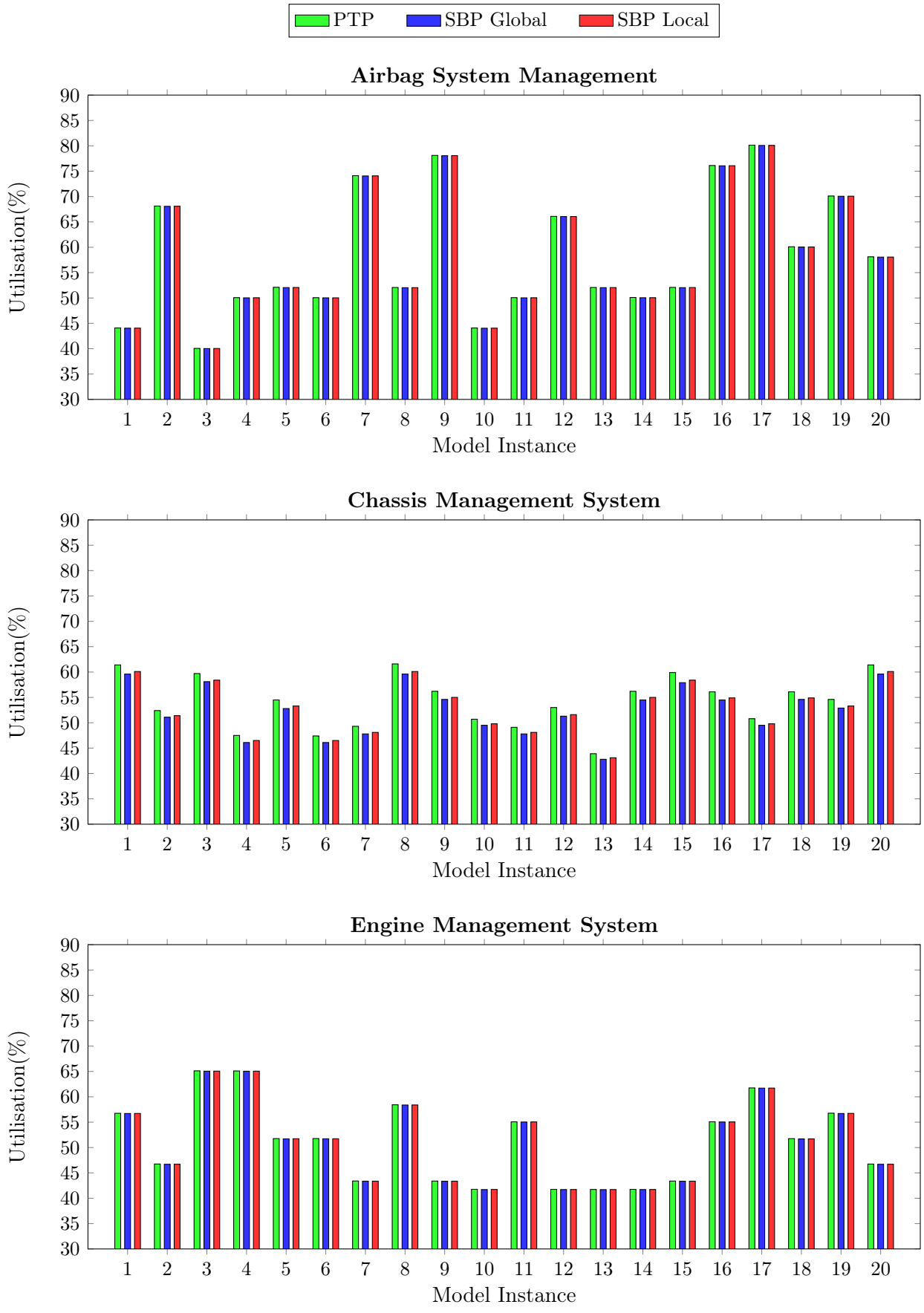


Figure 22: Average processor utilisation for 3 second runs of the synthetic models.

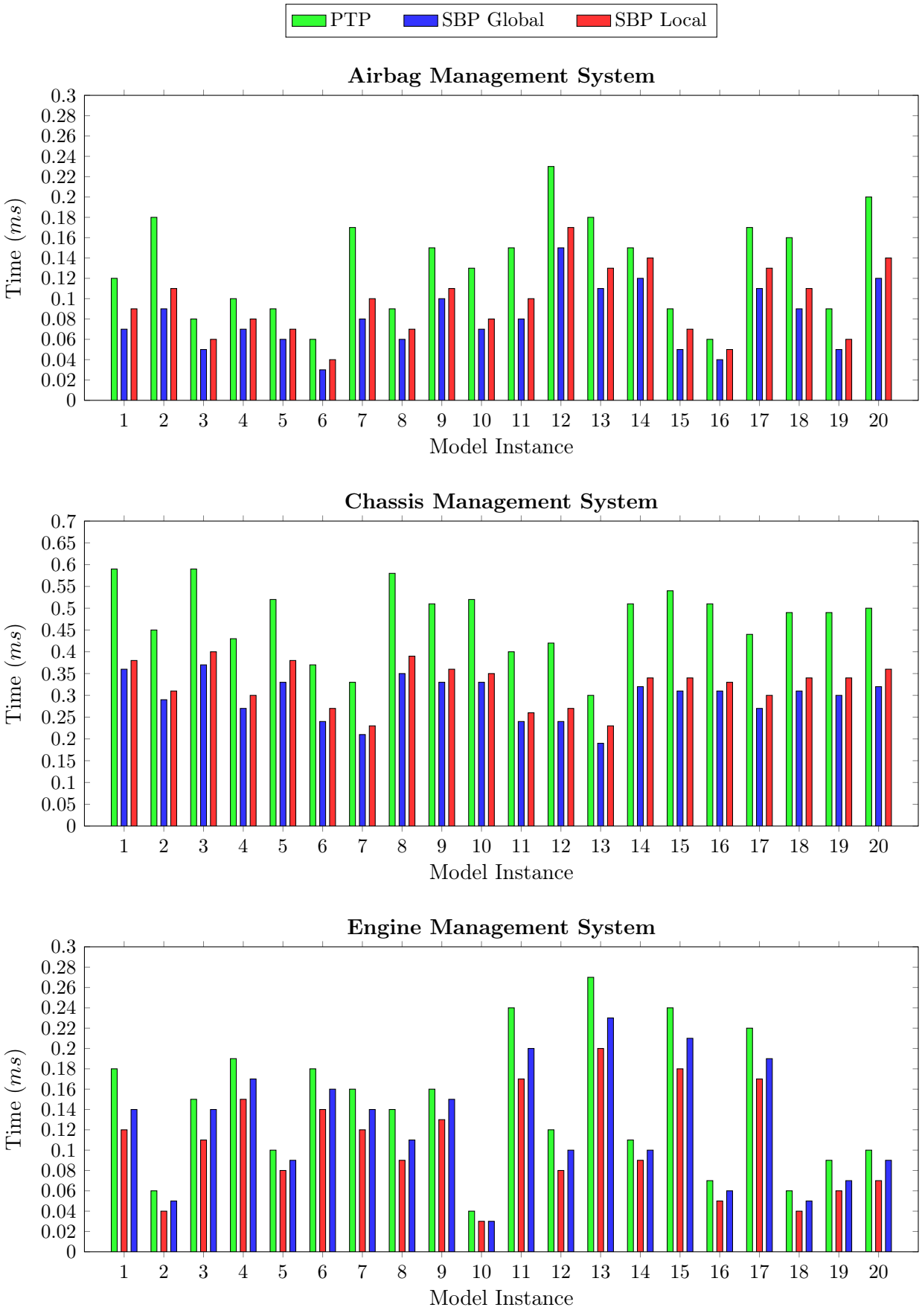


Figure 23: Total signal access times for the synthetic models.

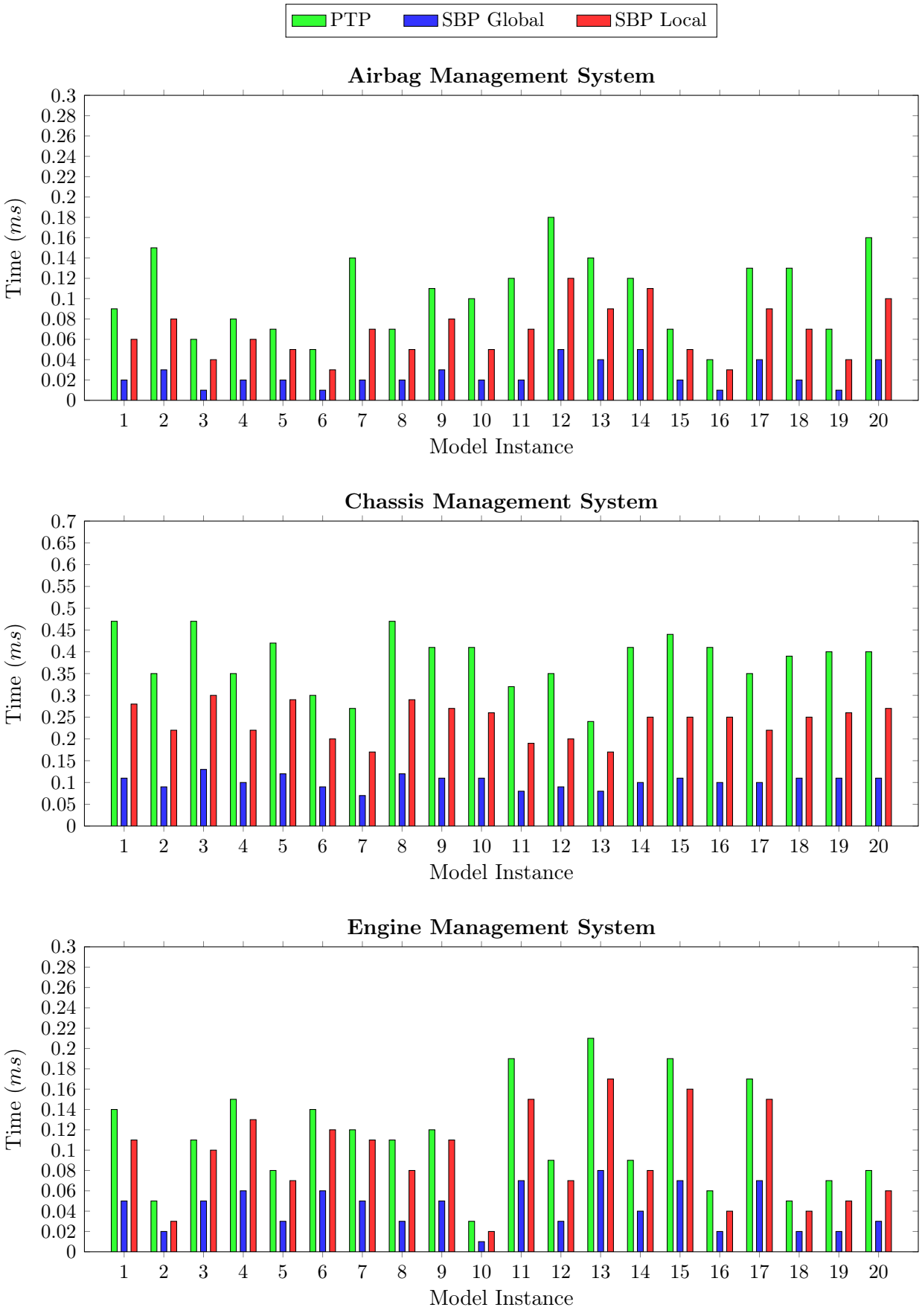


Figure 24: Total buffer management times for the synthetic models.

Table 13: FMTV task parameters (modified for LET). Hyper-period of $1,000$ ms

Task	**Period** (ms)	**LET** (ms)	**Observed Utilisation (%)**	**Core**
task1ms	1	1	30.68	0
task2ms	2	2	15.26	1
task5ms	5	5	12.92	0
task10ms	10	10	53.67	2
task20ms	20	20	31.21	1
task50ms	50	50	4.10	1
task100ms	100	100	6.68	1
task200ms	200	200	0.06	1
task1000ms	1,000	1,000	0.13	1

11 FMTV Case Study

The relevance of the implemented buffering optimisations (see Section 9.2) for an industrial AUTOSAR design is investigated in this section. We use the engine management system from the FMTV challenge [HDK+17], provided by Robert Bosch GmbH as an industrial case study to the research community. However, two main problems arose during benchmarking. First, one of the tasks has a WCET of 11.7 ms, which is longer than its 10 ms period. Second, the worst-case utilisation of three of the four cores exceeds 100%. Interestingly, no tasks executed beyond its period when 10 seconds of run-time is simulated because the worst-case is not reached. These two problems have also been reported by other FMTV challenge contestants [BPBN17]. We make the FMTV model schedulable by randomly reducing the instructions from the tasks with the highest load. Because we are interested in converting the periodic tasks to LET tasks, we assume that the interrupt service routines and sporadic tasks can be scheduled separately on their own core. The modified FMTV model has 9 LET tasks, $1,057$ runnables, and $8,824$ signals of which $1,248$ signals have no readers, 326 signals have no writers, and $3,364$ signals have constant values. The runnables access the signals directly, without any data protection mechanisms. The LET tasks are allocated over three cores and their parameters are provided in Table 13. The observed task utilisation is based on a simulation run of the model and does not represent the worst-case utilisation. The initial and activation offsets of the LET tasks are equal to zero.

Using the workflow of the synthetic benchmarks (see Section 10.1), three additional models are generated: a PTP buffered model, an SBP global buffered model, and an SBP local buffered model. The hardware platform used in the FMTV model is a single processor with four homogeneous cores at a fixed execution speed of 200 MHz, similar to Figure 1d. Each core has access to its own local memory module via a fast local bus, and access to a shared global memory module via a slower bus. All memory modules have a data width of 32 bits. The signal-to-memory module allocations are identical to those in the original FMTV model.

11.1 Preliminary Results

This section uses the evaluation metrics from Section 9.3 to compare the unbuffered and buffered FMTV models.

Total signal and buffer element count. Figure 25 shows the number of signals and buffer elements in the unbuffered and buffered FMTV models. The unbuffered model has 8,824 signals, and the PTP buffered model has nearly the same number for its buffers. However, SBP buffering requires a significant number of buffer indexes, drawn on top of the SBP buffering results in Figure 25. Thus, the SBP buffered models are more complex to implement than the PTP buffered model.

Total signal and buffer memory. Taking signal data sizes into account, Figure 26 shows the total memory that is needed for the signals and buffers in the unbuffered and buffered FMTV models. It is clear that the unbuffered model needs the least amount of memory for its signals, while the PTP buffered model needs nearly the same amount for its buffers. Ignoring the local variables and buffer indexes, less memory is needed for the SBP buffered models because of the applied buffering optimisations and that PTP buffering creates a buffer for each signal that a task accesses. The total memory needed for buffer indexes, each being 8 bits in size, are drawn on top of the SBP buffering results in Figure 26. The actual memory needed at run-time is less because the buffer indexes are allocated to the task stacks, so their occupied memory is freed when tasks terminate. The same reasoning applies to the local variables in the SBP local buffering model.

Processor utilisation. Figure 27 shows the average processor utilisation for 3 seconds of simulated run-time for the unbuffered and buffered FMTV models. With the unbuffered model as the baseline, the SBP global buffered model has a slight increase in processor utilisation. The SBP local buffered model has a slightly higher increase in processor utilisation because tasks have to write their local outputs to the signals. The PTP buffered model has the highest increase in processor utilisation, of 2%, because of signal reads and writes in the LET start and end routines.

Total signal access time. Figure 28 shows that more time is needed to complete all signal accesses with PTP buffering than with SBP buffering. As observed in the synthetic benchmarking, the sum of the signal access times in the LET start and end routines for PTP buffering contribute greatly towards the total signal access time.

Total buffer management time. Figure 29 shows that PTP buffering incurs the highest buffer management overhead, followed by SBP local buffering. The two observations identified in the synthetic benchmarking also apply here: (1) the alignment of multiple LET start and end routines could cause high bus contention for the PTP buffering model, and (2) the SBP local buffering model has to perform additional signal writes.

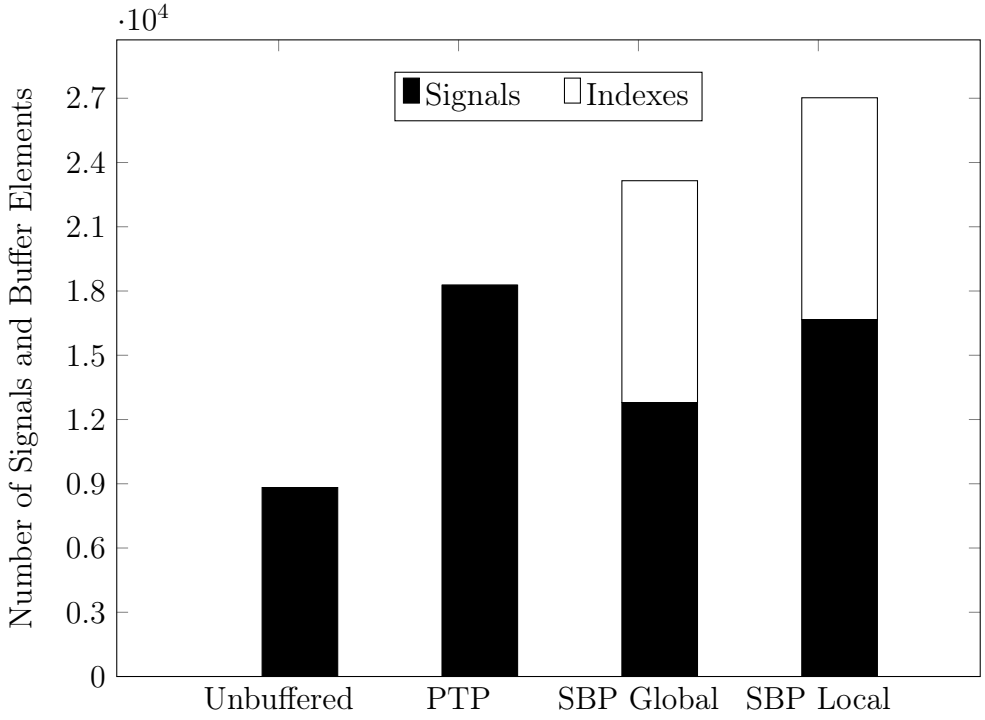


Figure 25: Total number of signals, buffer elements, and buffer indexes in the FMTV models.

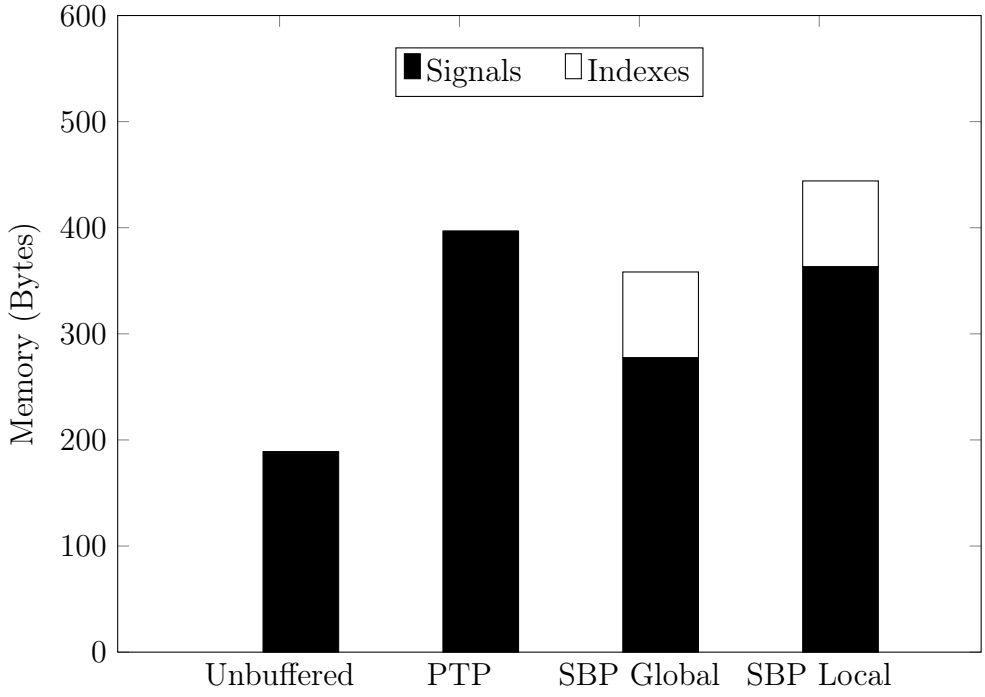


Figure 26: Total memory needed for the signals and buffers in the FMTV models.

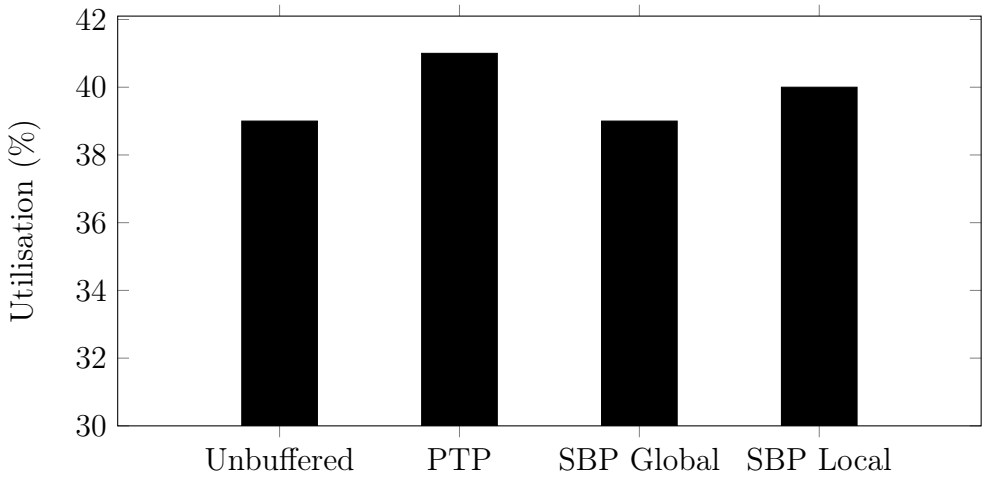


Figure 27: Average processor utilisation for 3 s of simulated run-time for the FMTV models.

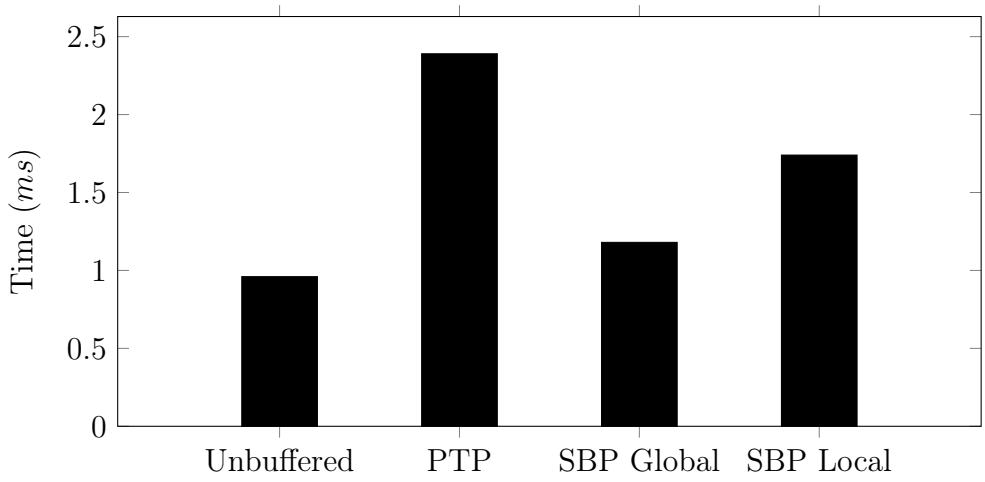


Figure 28: Total signal access time for the FMTV models.

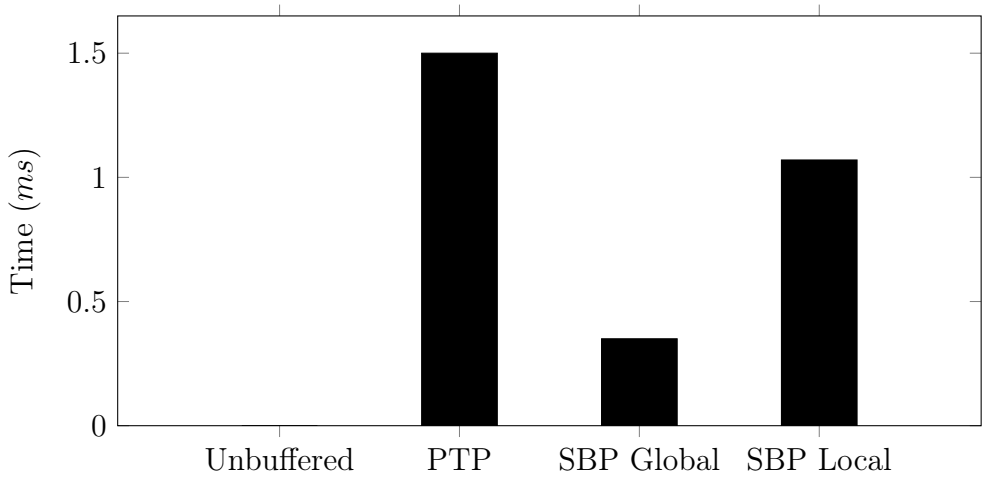


Figure 29: Total buffer management time for the FMTV models.

12 Conclusions

The *logical execution time* (LET) task model [KS12] presents an interesting solution to the mutli-core challenges that the automotive industry is facing. The need to precisely define the timing of task communications enables time-predictable and deterministic execution behaviours that are platform agnostic. However, the LET communication model must be implemented in a semantics preserving manner, otherwise its benefits are lost. Central to this is the need to buffer communication signals when tasks are unable to synchronise at the same time.

We have adapted and improved on the dynamic buffering protocol [STC06] for statically scheduled LET-based tasks. A range of algorithms and heuristics were developed by us to reduce the buffer memory, processor utilisation, and end-to-end response times of LET-based multi-core AUTOSAR designs [HvHM+16, RNH+15]: (1) Memory-efficient SBP buffering schedules are generated from logical task schedules. (2) The allocation of tasks-to-cores and signal buffers-to-memory modules and the selection of PTP or SBP buffering for each signal are decided by an MILP formulation or genetic algorithm. (3) A physical task schedule is generated from task-to-core allocations and scheduling hints. The task schedule could be scaled down to improve the system's overall end-to-end response times. (4) The original SBP buffering schedules are refined with the task execution times from a physical task schedule. (5) SBP buffers are merged and the final deployment is generated. Offline, rather than online, optimisations were preferred because they introduce the least amount of run-time uncertainty, which is desirable when designing hard real-time automotive systems with strict timing constraints.

The design phase optimisations and a simplified SBP buffer merging heuristic were prototyped into the TA Tool Suite [Vec18] and evaluated. Synthetic benchmarks with parameters based on actual airbag, chassis, and engine management systems were used, along with one industrial example from the FMTV challenge. Preliminary results suggest that a LET-based system with SBP buffering uses less memory and processor utilisation than with PTP buffering. For future work, the prototyping of the remaining algorithms and heuristics is required, along with further benchmarking and evaluations. We hope to improve the output of the proposed algorithms and heuristics as more is learnt about system characteristics in the evaluations.

References

[AUT17a] AUTOSAR. Release 4.3.1. Available at http://www.autosar.org, December 2017. Last accessed August 2018.

[AUT17b] AUTOSAR. Specification of RTE, December 2017. Release 4.3.1. Available at http://www.autosar.org. Last accessed August 2018.

[AUT17c] AUTOSAR. Specification of Timing Extensions, December 2017. Release 4.3.1. Available at http://www.autosar.org. Last accessed August 2018.

[BCB+08] Björn B. Brandenburg, John M. Calandrino, Aaron Block, Hennadiy Leontyev, and James H. Anderson. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 342–353, April 2008.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages 12 Years Later. *IEEE*, 91(1):64 – 83, January 2003.

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a Moving Window over Streaming Data. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634. Society for Industrial and Applied Mathematics, January 2002.

[BGG+71] M. Benichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in Mixed-Integer Linear Programming. *Mathematical Programming*, 1(1):76–94, December 1971.

[BK73] Coen Bron and Joep Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[BKU16] Christian Bradatsch, Florian Kluge, and Theo Ungerer. Data Age Diminution in the Logical Execution Time Model. In *Architecture of Computing Systems (ARCS)*, volume 9637 of *Lecture Notes in Computer Science*, pages 173–184. Springer, March 2016.

[BPBN17] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores. In *Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Inria, June 2017. Available at https://www.ecrts.org/forum/viewtopic.php?f=32&t=87. Last accessed August 2018.

[CM05] Paul Caspi and Oded Maler. From Control Loops to Real-Time Programs. In *Handbook of Networked and Embedded Control Systems*, pages 395–418. Birkhäuser Boston, 2005.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[EKQS18] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports*, 8(2):122–149, August 2018.

[FFPT05] Emilia Farcas, Claudiu Farcas, Wolfgang Pree, and Josef Templ. Transparent Distribution of Real-time Components Based on Logical Execution Time. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 31–39. ACM, July 2005.

[FLSN14] Hamid Reza Faragardi, Björn Lisper, Kristian Sandström, and Thomas Nolte. An Efficient Scheduling of AUTOSAR Runnables to Minimize Communication Cost in Multi-Core Systems. In *Telecommunications (IST)*, pages 41–48. IEEE, September 2014.

[FM06] Eugene C. Freuder and Alan K. Mackworth. Chapter 2 - Constraint Satisfaction: An Emerging Paradigm. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 13–27. Elsevier, March 2006.

[FNG+09] Alberto Ferrari, Marco Di Natale, Giacomo Gentile, Giovanni Reggiani, and Paolo Gai. Time and Memory Tradeoffs in the Implementation of AUTOSAR Components. In *Design, Automation Test in Europe Conference Exhibition*, pages 864–869. European Design and Automation Association, April 2009.

[GGL14] Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Timing Analysis of Parallel Software Using Abstract Execution. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 8318 of *Lecture Notes in Computer Science*, pages 59–77. Springer, January 2014.

[GKKL16] Alex Gavryushkin, Bakhadyr Khoussainov, Mikhail Kokho, and Jiamou Liu. Dynamic Algorithms for Multimachine Interval Scheduling Through Analysis of Idle Intervals. *Algorithmica*, 76(4):1160–1180, December 2016.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[Hab72] Arie Nico Habermann. Synchronization of Communicating Processes. *Communications of the ACM*, 15(3):171–176, March 1972.

[HDK+17] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. WATERS Industrial Challenge 2017. In *Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Inria, June 2017. Available at https://waters2017.inria.fr/challenge. Last accessed August 2018.

[Her90] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 197–206. ACM, November 1990.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 166–184. Springer, October 2001.

[HK07] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-time Code. *ACM Transactions on Programming Languages and Systems*, 29(6):33:1–33:29, October 2007.

[HvHM+16] Julien Hennig, Hermann von Hasseln, Hassan Mohammad, Stefan Resmerita, Stefan Lukesch, and Andreas Naderlinger. Towards Parallelizing Legacy Embedded Control Software Using the LET Programming Paradigm. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, April 2016.

[HZN+14] Gang Han, Haibo Zeng, Marco Di Natale, Xue Liu, and Wenhua Dou. Experimental Evaluation and Selection of Data Consistency Mechanisms for Hard Real-Time Applications on Multicore Platforms. *IEEE Transactions on Industrial Informatics*, 10(2):903–918, May 2014.

[Inf14] Infineon. *AURIX^{TM} TC27x C-Step 32-bit Single-Chip Microcontroller: User's Manual*. Infineon Technologies AG, Munich, Germany, 2.2 edition, December 2014.

[Inf17] Information Technology for European Advancement (ITEA2). AMALTHEA. http://www.amalthea-project.org, August 2017. Last accessed September 2018.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer, March 1972.

[KKTM10] Kay Klobedanz, Christoph Kuznik, Andreas Thuy, and Wolfgang Müller. Timing Modeling and Analysis for AUTOSAR-based Software Development–A Case Study. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 642–645. IEEE, March 2010.

[Kop91] Hermann Kopetz. Event-Triggered Versus Time-Triggered Real-Time Systems. In *Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101. Springer, July 1991.

[KQBS15] Sebastian Kehr, Eduardo Quiñones, Bert Böddeker, and Günter Schäfer. Parallel Execution of AUTOSAR Legacy Applications on Multicore ECUs with Timed Implicit Communication. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, June 2015.

[KR93] Hermann Kopetz and Johannes Reisinger. The Non-Blocking Write Protocol NBW: A solution to a Real-Time Synchronization Problem. In *Real-Time Systems Symposium (RTSS)*, pages 131–137. IEEE, December 1993.

[KS12] Christoph M. Kirsch and Ana Sokolova. The Logical Execution Time Paradigm. In *Advances in Real-Time Systems (ARTS)*, chapter 5, pages 103–120. Springer, 2012.

[KSU83] Florian Kluge, Martin Schoeberl, and Theo Ungerer. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[KSU16] Florian Kluge, Martin Schoeberl, and Theo Ungerer. Support for the Logical Execution Time Model on a Time-predictable Multicore Processor. *SIGBED Review–Special Issue on International Workshop on RealTime Networks (RTN)*, 13(4):61–66, November 2016.

[LL73] Chung Laung Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., 1st edition, June 2012.

[NWV08] Marco Di Natale, Guoqiang Wang, and Alberto Sangiovanni Vincentelli. Optimizing the Implementation of Communication in Synchronous Reactive Models. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 169–179. IEEE, April 2008.

[OSE05] OSEK/VDX. Operating Sytem, February 2005. Specification 2.3.3. Available at http://www.osek-vdx.org. Last accessed November 2017.

[OSHK09] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. From a Federated to an Integrated Automotive Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):956–965, July 2009.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., 1982.

[Ray13] Michel Raynal. Solving Mutual Exclusion. In *Concurrent Programming: Algorithms, Principles, and Foundations*, chapter 2, pages 15–60. Springer, 2013.

[RNH+15] Stefan Resmerita, Andreas Naderlinger, Manuel Huber, Kenneth Butts, and Wolfgang Pree. Applying Real-Time Programming to Legacy Embedded Control Software. In *Real-Time Distributed Computing (ISORC)*, pages 1–8. IEEE, April 2015.

[RNL17] Stefan Resmerita, Andreas Naderlinger, and Stefan Lukesch. Efficient Realization of Logical Execution Times in Legacy Embedded Software. In *Formal Methods and Models for System Design (MEMOCODE)*, pages 36–45. ACM, September 2017.

[SCCM15] Salah Eddine Saidi, Sylvain Cotard, Khaled Chaaban, and Kevin Marteil. An ILP Approach for Mapping AUTOSAR Runnables on Multi-core Architectures. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, pages 6:1–6:8. ACM, January 2015.

[ST00] H. Sundell and P. Tsigas. Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 433–440. IEEE, December 2000.

[STC06] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A Memory-Optimal Buffering Protocol for Preservation of Synchronous Semantics Under Preemptive Scheduling. In *Embedded Software (EMSOFT)*, pages 21–33. ACM, October 2006.

[Vec18] Vector Informatik GmbH. Vector Academic and Research License Program. https://vector.com/vi_embedded_timing-architecture_de.html, April 2018. Last accessed September 2018.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-time Problem–Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

[WMM+13] Ernest Wozniak, Asma Mehiaoui, Chokri Mraidha, Sara Tucci-Piergiovanni, and Sébastien Gerard. An Optimization Approach for the Synthesis of AUTOSAR Architectures. In *Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–10. IEEE, September 2013.

[WNSV07] Guoqiang Wang, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protocols. Technical Report UCB/EECS-2007-81, EECS Department, University of California, Berkeley, June 2007.

[WNSV10] Guoqiang Wang, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Optimal Synthesis of Communication Procedures in Real-Time Synchronous Reactive Models. *IEEE Transactions on Industrial Informatics*, 6(4):729–743, November 2010.

[WS99] Yun Wang and Manas Saksena. Scheduling Fixed-Priority Tasks with Preemption Threshold. In *Real-Time Computing Systems and Applications (RTCSA)*, pages 328–335. IEEE, December 1999.

[YKRB14] Eugene Yip, Matthew M. Y. Kuo, Partha S. Roop, and David Broman. Relaxing the Synchronous Approach for Mixed-Criticality Systems. In *IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, pages 89–100. IEEE, April 2014.

[ZN12] Haibo Zeng and Marco Di Natale. Efficient Implementation of AUTOSAR Components with Minimal Memory Usage. In *Industrial Embedded Systems (SIES)*, pages 130–137. IEEE, June 2012.

[ZNZ14] Haibo Zeng, Marco Di Natale, and Qi Zhu. Minimizing Stack and Communication Memory Usage in Real-Time Embedded Applications. *ACM Transactions on Embedded Computing Systems*, 13(5s):149:1–149:25, July 2014.

Bamberger Beiträge zur Wirtschaftsinformatik

Nr. 1 (1989)	Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990)
Nr. 2 (1990)	Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG
Nr. 3 (1990)	Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen
Nr. 4 (1990)	Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990)
Nr. 5 (1990)	Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)
Nr. 6 (1991)	Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten
Nr. 7 (1991)	Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender
Nr. 8 (1991)	Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung
Nr. 9 (1992)	Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell
Nr. 10 (1992)	Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)
Nr. 11 (1992)	Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells
Nr. 12 (1992)	Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt
Nr. 13 (1992)	Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell
Nr. 14 (1992)	Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen
Nr. 15 (1992)	Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch
Nr. 16 (1992)	Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip
Nr. 17 (1993)	Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation
Nr. 18 (1993)	Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells
Nr. 19 (1994)	Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach

Nr. 20 (1994) Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994

Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach -. 2nd edition, November 1994

Nr. 21 (1994) Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen

Nr. 22 (1994) Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems

Nr. 23 (1994) Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle

Nr. 24 (1994) Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen

Nr. 25 (1994) Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme

Nr. 26 (1995) Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes

Nr. 27 (1995) Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995

Nr. 28 (1995) Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach

Nr. 30 (1995) Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit

Nr. 31 (1995) Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse

Nr. 32 (1995) Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger

Nr. 33 (1995) Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen?

Nr. 34 (1995) Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -

Nr. 35 (1995) Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme

Nr. 36 (1996) Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996

Nr. 37 (1996) Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996

Nr. 38 (1996) Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996

Nr. 39 (1996) Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze

Nr. 40 (1997) Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997

Nr. 41 (1997) Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997

Nr. 42 (1997) Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997

Nr. 43 (1997): Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997

Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998

Nr. 44 (1997) Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung

Nr. 45 (1998) Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998

Nr. 46 (1998) Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997

Nr. 47 (1998) Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998

Nr. 48 (1998) Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering

Nr. 49 (1998) Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460

Nr. 50 (1999) Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes)

Nr. 51 (1999) Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science)

Nr. 52 (1999) Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule" im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999

Nr. 53 (1999) Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999

Nr. 54 (1999) Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation.

Nr. 55 (2000) Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur

Nr. 56 (2000) Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000

Nr. 57 (2000) Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models.

Nr. 58 (2000) Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten.

Nr. 59 (2001) Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001

Nr. 60 (2001) Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001" im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61

Note: The title of our technical report series has been changed from *Bamberger Beiträge zur Wirtschaftsinformatik* to *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* starting with TR No. 61

Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik

Nr. 61 (2002) Goré R., Mendler M., de Paiva V. (Hrsg.): Proceedings of the International Workshop on Intuitionistic Modal Logic and Applications (IMLA 2002), Copenhagen, July 2002.

Nr. 62 (2002) Sinz E.J., Plaha M., Ulbrich-vom Ende A.: Datenschutz und Datensicherheit in einem landesweiten Data-Warehouse-System für das Hochschulwesen. Erscheint in: Beiträge zur Hochschulforschung, Heft 4-2002, Bayerisches Staatsinstitut für Hochschulforschung und Hochschulplanung, München 2002

Nr. 63 (2005) Aguado, J., Mendler, M.: Constructive Semantics for Instantaneous Reactions

Nr. 64 (2005) Ferstl, O.K.: Lebenslanges Lernen und virtuelle Lehre: globale und lokale Verbesserungspotenziale. Erschienen in: Kerres, Michael; Keil-Slawik, Reinhard (Hrsg.); Hochschulen im digitalen Zeitalter: Innovationspotenziale und Strukturwandel, S. 247 – 263; Reihe education quality forum, herausgegeben durch das Centrum für eCompetence in Hochschulen NRW, Band 2, Münster/New York/München/Berlin: Waxmann 2005

Nr. 65 (2006) Schönberger, Andreas: Modelling and Validating Business Collaborations: A Case Study on RosettaNet

Nr. 66 (2006) Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, Karsten Loesing, and Guido Wirtz: Concealing Presence Information in Instant Messaging Systems, April 2006

Nr. 67 (2006) Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz: Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems, April 2006

Nr. 68 (2006) Michael Mendler, Thomas R. Shiple, Gérard Berry: Constructive Circuits and the Exactness of Ternary Simulation

Nr. 69 (2007) Sebastian Hudert: A Proposal for a Web Services Agreement Negotiation Protocol Framework . February 2007

Nr. 70 (2007) Thomas Meins: Integration eines allgemeinen Service-Centers für PC-und Medientechnik an der Universität Bamberg – Analyse und Realisierungs-Szenarien. February 2007 (out of print)

Nr. 71 (2007) Andreas Grünert: Life-cycle assistance capabilities of cooperating Software Agents for Virtual Enterprises. März 2007

Nr. 72 (2007) Michael Mendler, Gerald Lüttgen: Is Observational Congruence on μ-Expressions Axiomatisable in Equational Horn Logic?

Nr. 73 (2007) Martin Schissler: out of print

Nr. 74 (2007) Sven Kaffille, Karsten Loesing: Open chord version 1.0.4 User's Manual. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 74, Bamberg University, October 2007. ISSN 0937-3349.

Nr. 75 (2008) Karsten Loesing (Hrsg.): Extended Abstracts of the Second *Privacy Enhancing Technologies Convention* (PET-CON 2008.1). Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 75, Bamberg University, April 2008. ISSN 0937-3349.

Nr. 76 (2008) Gregor Scheithauer, Guido Wirtz: Applying Business Process Management Systems – A Case Study. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 76, Bamberg University, May 2008. ISSN 0937-3349.

Nr. 77 (2008) Michael Mendler, Stephan Scheele: Towards Constructive Description Logics for Abstraction and Refinement. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 77, Bamberg University, September 2008. ISSN 0937-3349.

Nr. 78 (2008) Gregor Scheithauer, Matthias Winkler: A Service Description Framework for Service Ecosystems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 78, Bamberg University, October 2008. ISSN 0937-3349.

Nr. 79 (2008) Christian Wilms: Improving the Tor Hidden Service Protocol Aiming at Better Performances. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 79, Bamberg University, November 2008. ISSN 0937-3349.

Nr. 80 (2009) Thomas Benker, Stefan Fritzemeier, Matthias Geiger, Simon Harrer, Tristan Kessner, Johannes Schwalb, Andreas Schönberger, Guido Wirtz: QoS Enabled B2B Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 80, Bamberg University, May 2009. ISSN 0937-3349.

Nr. 81 (2009) Ute Schmid, Emanuel Kitzelmann, Rinus Plasmeijer (Eds.): Proceedings of the ACM SIGPLAN Workshop on *Approaches and Applications of Inductive Programming* (AAIP'09), affiliated with ICFP 2009, Edinburgh, Scotland, September 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 81, Bamberg University, September 2009. ISSN 0937-3349.

Nr. 82 (2009) Ute Schmid, Marco Ragni, Markus Knauff (Eds.): Proceedings of the KI 2009 Workshop *Complex Cognition*, Paderborn, Germany, September 15, 2009. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 82, Bamberg University, October 2009. ISSN 0937-3349.

Nr. 83 (2009) Andreas Schönberger, Christian Wilms and Guido Wirtz: A Requirements Analysis of Business-to-Business Integration. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 83, Bamberg University, December 2009. ISSN 0937-3349.

Nr. 84 (2010) Werner Zirkel, Guido Wirtz: A Process for Identifying Predictive Correlation Patterns in Service Management Systems. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 84, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 85 (2010) Jan Tobias Mühlberg und Gerald Lüttgen: Symbolic Object Code Analysis. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 85, Bamberg University, February 2010. ISSN 0937-3349.

Nr. 86 (2010) Werner Zirkel, Guido Wirtz: Proaktives Problem Management durch Eventkorrelation – ein *Best Practice* Ansatz. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 86, Bamberg University, August 2010. ISSN 0937-3349.

Nr. 87 (2010) Johannes Schwalb, Andreas Schönberger: Analyzing the Interoperability of WS-Security and WS-ReliableMessaging Implementations. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 87, Bamberg University, September 2010. ISSN 0937-3349.

Nr. 88 (2011) Jörg Lenhard: A Pattern-based Analysis of WS-BPEL and Windows Workflow. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 88, Bamberg University, March 2011. ISSN 0937-3349.

Nr. 89 (2011) Andreas Henrich, Christoph Schlieder, Ute Schmid [eds.]: Visibility in Information Spaces and in Geographic Environments – Post-Proceedings of the KI'11 Workshop. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 89, Bamberg University, December 2011. ISSN 0937-3349.

Nr. 90 (2012) Simon Harrer, Jörg Lenhard: Betsy - A BPEL Engine Test System. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 90, Bamberg University, July 2012. ISSN 0937-3349.

Nr. 91 (2013) Michael Mendler, Stephan Scheele: On the Computational Interpretation of CKn for Contextual Information Processing - Ancillary Material. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 91, Bamberg University, May 2013. ISSN 0937-3349.

Nr. 92 (2013) Matthias Geiger: BPMN 2.0 Process Model Serialization Constraints. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 92, Bamberg University, May 2013. ISSN 0937-3349.

Nr. 93 (2014) Cedric Röck, Simon Harrer: Literature Survey of Performance Benchmarking Approaches of BPEL Engines. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 93, Bamberg University, May 2014. ISSN 0937-3349.

Nr. 94 (2014) Joaquin Aguado, Michael Mendler, Reinhard von Hanxleden, Insa Fuhrmann: Grounding Synchronous Deterministic Concurrency in Sequential Programming. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 94, Bamberg University, August 2014. ISSN 0937-3349.

Nr. 95 (2014) Michael Mendler, Bruno Bodin, Partha S Roop, Jia Jie Wang: WCRT for Synchronous Programs: Studying the Tick Alignment Problem. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 95, Bamberg University, August 2014. ISSN 0937-3349.

Nr. 96 (2015) Joaquin Aguado, Michael Mendler, Reinhard von Hanxleden, Insa Fuhrmann: Denotational Fixed-Point Semantics for Constructive Scheduling of Synchronous Concurrency. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 96, Bamberg University, April 2015. ISSN 0937-3349.

Nr. 97 (2015) Thomas Benker: Konzeption einer Komponentenarchitektur für prozessorientierte OLTP- & OLAP-Anwendungssysteme. Bamberger Beiträge zur Wirtschafts-informatik und Angewandten Informatik Nr. 97, Bamberg University, Oktober 2015. ISSN 0937-3349.

Nr. 98 (2016) Sascha Fendrich, Gerald Lüttgen: A Generalised Theory of Interface Automata, Component Compatibility and Error. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 98, Bamberg University, March 2016. ISSN 0937-3349.

Nr. 99 (2014) Christian Preißinger, Simon Harrer: Static Analysis Rules of the BPEL Specification: Tagging, Formalization and Tests. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 99, Bamberg University, August 2014. ISSN 0937-3349.

Nr. 100 (2016) Cedrik Röck, Stefan Kolb: Nucleus - Unified Deployment and Management for Platform as a Service. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 100, Bamberg University, March 2016. ISSN 0937-3349.

Nr. 101 (2016) Michael Mendler, Partha S. Roop, Bruno Bodin: A Novel WCET Semantics of Synchronous Programs. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 101, Bamberg University, June 2016. ISSN 0937-3349.

Nr. 102 (2017) Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha Roop, Reinhard von Hanxleden: Clock-Synchronised Shared Objects for Deterministic Concurrency. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 102, Bamberg University, July 2017. ISSN 0937-3349.

Nr. 103 (2018) Eugene Yip, Erjola Lalo, Gerald Lüttgen, Michael Deubzer, Andreas Sailer: Optimized Buffering of Time-Triggered Automotive Software. Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik Nr. 103, Bamberg University, September 2018. ISSN 0937-3349.