Optimized Buffering for Time-Triggered Automotive Software

Illustrated Overview of the Developed Algorithms and Heuristics

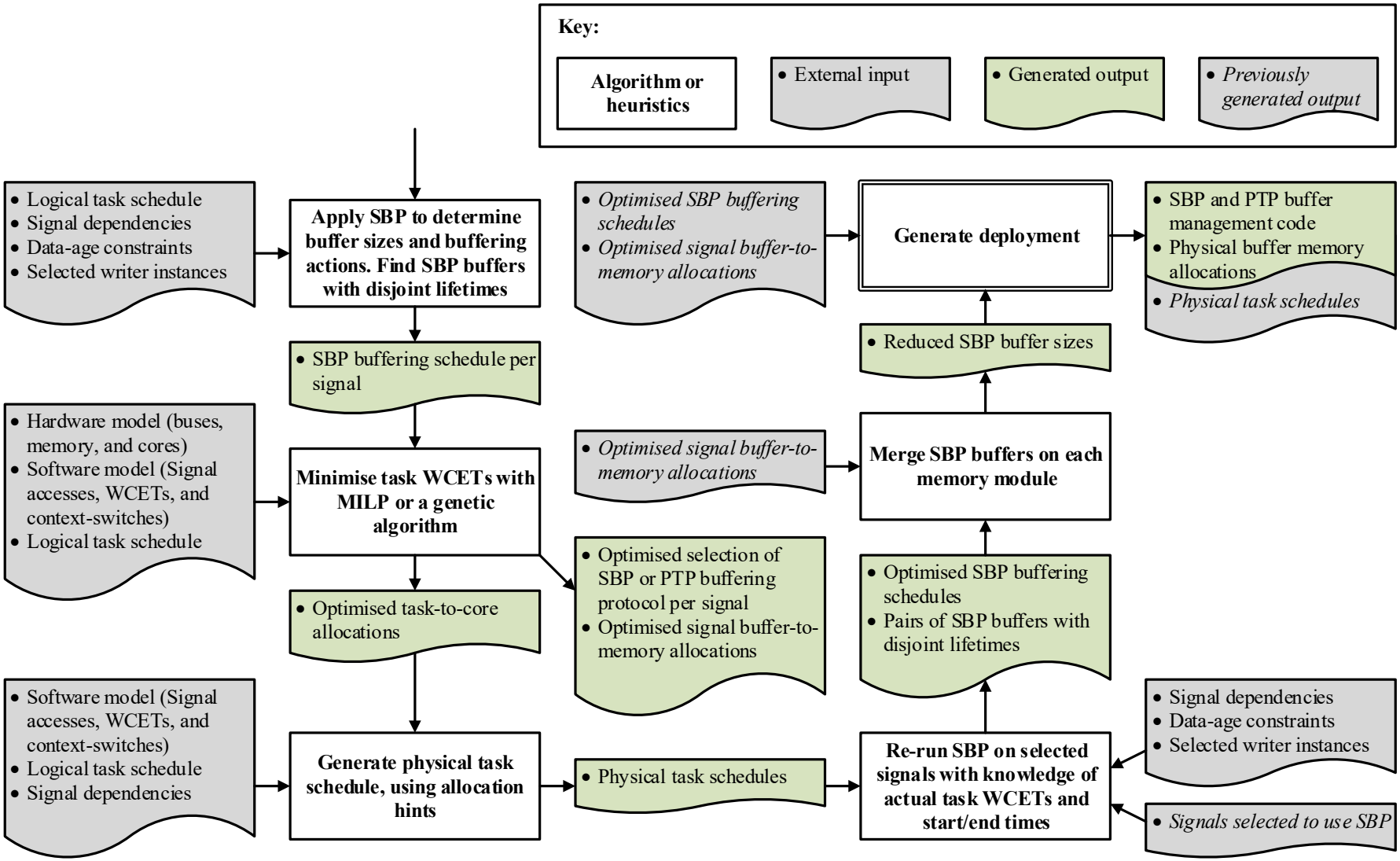
Contents

- Overview of the design and deployment optimisations
- Design level optimisations
- Deployment level optimisations with MILP
- Deployment level optimisations with a genetic algorithm

Overview

- Optimisation goals
  - Minimise buffer sizes, frequency of buffer writes, and allocated time for task execution
  - While maintaining schedulability, and data-flow and timing of the original LET design
- Design level optimisations
  - Platform independent optimisations that remain sound at the deployment level(s).
  - Only considers LET start and end times. Ignores resource allocations
- Deployment level optimisations
  - Platform dependent optimisations based on specific knowledge of the available resources
  - Considers resource allocations, e.g., task-to-core allocations, buffer-to-memory allocations, execution time allocations, and run-time overheads

Optimisation and Heuristics Overview



Overview

- The physical task and buffering schedules shall be schedulable
  - The utilisation of each core and memory module shall not exceed 100%
  - The data-flow and timing of the original design shall be preserved by the deployment
- Pessimistic approach to centralised buffering
  - Assume all signal writes and reads require buffering
  - Prune away unnecessary use of buffers in a stepwise manner

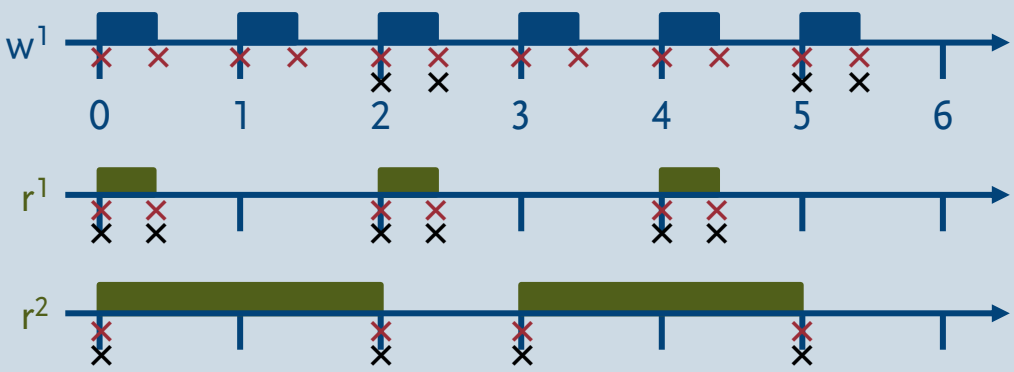
Informal Description: Design Level

Intuition: DesignOptimisations(...) Function

1. For each signal, find all the time points that are of interest to the static buffering protocol (SBP). These are:
   - the earliest and latest times that reader instances can read from the signal: Their LET boundaries
   - the earliest and latest times that writer instances can write to the signal
     - For the global programming style (intermediate signal values are written to the buffer): The LET boundaries of all writer instances
     - For the local programming style (only final signal values are written to the buffer): The LET boundaries of some writer instances (GetWriterUses function)
2. Run the SBP function over the time points to get the buffering schedule and lifetime
3. Find pairs of signal buffers with disjoint lifetimes, which can be merged during deployment (GetDisjBuffs function)

Intuition: DesignOptimisations(...) Function

Example



Data-age constraints:

$w^1 \xrightarrow{s,3} r^1$

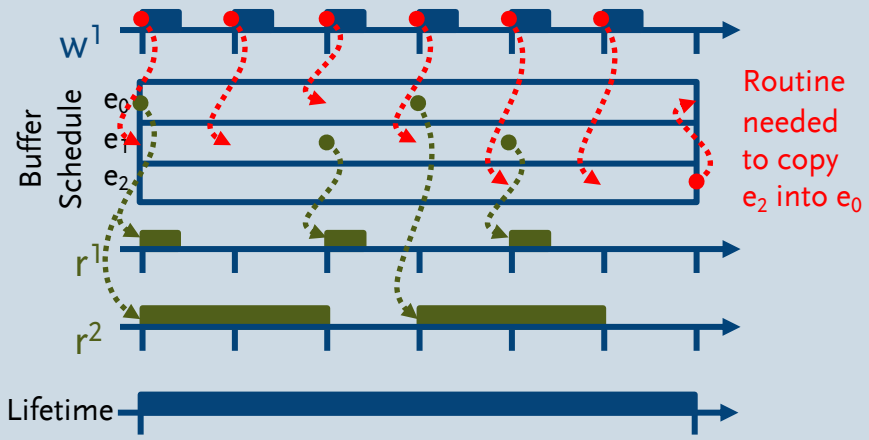
$w^1 \xrightarrow{s,3} r^2$

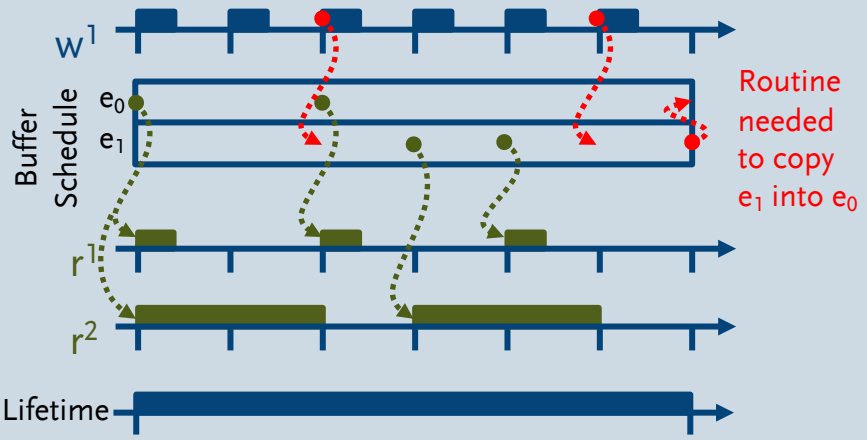
× = Time points of interest for the global programming style

× = Time points of interest for the local programming style

SBP for the global style

SBP for the local style





Intuition: GetWriterUses(...) Function

Helper function that selects a subset of all writer instances that all reader instances can read from

- Equivalent to the "set cover problem"
- Universe $\mathcal{U} = \{r_0^1, r_1^1, r_2^1, r_0^2, r_1^2\}$
- Sets $\mathcal{S} = \left\{\{r_0^1, r_1^1, r_0^2\}_{-1}, \{r_1^1, r_1^2\}_0, \{r_1^1, r_2^1, r_1^2\}_1, \{r_2^1, r_1^2\}_2, \{r_2^1\}_3\right\}$ which have been derived from the following table:

Reader instance	Writer w^1 instances				
	-1	0	1	2	3
r_0^1	✓				
r_1^1	✓	✓	✓		
r_2^1			✓	✓	✓
r_0^2	✓				
r_1^2		✓	✓	✓	

- The goal is to find the minimum $\mathcal{C} \subseteq \mathcal{S}$ that contains/covers $\mathcal{U}$, while choosing the latest possible writer instances (subscript of each set in $\mathcal{S}$)

Intuition: GetWriterUses(...) Function

Helper function that selects a subset of all writer instances that all reader instances can read from

1. Find all the writer instances that each reader instance can read from, considering their data-age constraints. Organise into a "potential" table
2. For each reader instance, find the latest possible writer instance that it can read from. Organise into a "latest" table

Represents the signal's initial value →

Potential Table

Writer instance	Reader instances
w_{-1}^1	$\{r_0^1, r_1^1, r_0^2\}$
w_0^1	$\{r_1^1, r_1^2\}$
w_1^1	$\{r_1^1, r_2^1, r_1^2\}$
w_2^1	$\{r_2^1, r_1^2\}$
w_3^1	$\{r_2^1\}$
w_4^1	$\{\}$
w_5^1	$\{\}$

Latest Table

Writer instance	Reader instances
w_{-1}^1	$\{r_0^1, r_0^2\}$
w_0^1	$\{\}$
w_1^1	$\{r_1^1\}$
w_2^1	$\{r_1^2\}$
w_3^1	$\{r_2^1\}$
w_4^1	$\{\}$
w_5^1	$\{\}$

Latest table gives an upper bound on the writer instances that are need

Intuition: GetWriterUses(...) Function

Helper function that selects a subset of all writer instances that all reader instances can read from

3. Iterate through the latest table and select the latest possible writer instance that needs to be read. Every time a writer instance is selected, record its corresponding reader instances

Potential Table

Writer instance	Reader instances
w_{-1}^1	$\{r_0^1, r_1^1, r_0^2\}$
w_0^1	$\{r_1^1, r_1^2\}$
w_1^1	$\{r_1^1, r_2^1, r_1^2\}$
w_2^1	$\{r_2^1, r_1^2\}$
w_3^1	$\{r_2^1\}$
w_4^1	$\{\}$
w_5^1	$\{\}$

Latest Table

Writer instance	Reader instances	Selected writer instance	Corresponding reader instances
w_{-1}^1	$\{r_0^1, r_0^2\}$	$\{w_{-1}^1\}$	$\{r_0^1, r_1^1, r_0^2\}$
w_0^1	$\{\}$	$\{w_{-1}^1\}$	$\{r_0^1, r_1^1, r_0^2\}$
w_1^1	$\{r_1^1\}$	$\{w_{-1}^1\}$	$\{r_0^1, r_1^1, r_0^2\}$
w_2^1	$\{r_1^2\}$	$\{w_{-1}^1, w_2^1\}$	$\{r_0^1, r_1^1, r_2^1, r_0^2, r_1^2\}$
w_3^1	$\{r_2^1\}$	$\{w_{-1}^1, w_2^1\}$	$\{r_0^1, r_1^1, r_2^1, r_0^2, r_1^2\}$
w_4^1	$\{\}$	$\{w_{-1}^1, w_2^1\}$	$\{r_0^1, r_1^1, r_2^1, r_0^2, r_1^2\}$
w_5^1	$\{\}$	$\{w_{-1}^1, w_2^1\}$	$\{r_0^1, r_1^1, r_2^1, r_0^2, r_1^2\}$

Intuition: SBP(...) Function

Process the time points in chronological order

1. Track each writer's `prev` and `next` pointers
2. Track the latest `prev` that all readers will read from
3. Track the buffer elements that are occupied
4. Create new buffer elements when the buffer is fully occupied
5. Track the buffer's lifetime. Contributions from:
   - the LET start of each writer instance to the LET end of each corresponding reader instance
   - all the writer instances, if the global programming style is used

Formal Description: Design Level

Definitions

LET task:

- $t^a = \langle period, letStart, letEnd \rangle \in T$: A LET task, which has a name "a", a period, a LET start time, and a LET end time

LET task instance

- $t_i^a = \langle periodStart, letStart, letEnd \rangle$: Instance $i \in \mathbb{N}_0$ of LET task t^a, where $t_i^a.periodStart = t^a.period \times i$, $t_i^a.letStart = t_i^a.periodStart + t^a.letStart$, and $t_i^a.letEnd = t_i^a.periodStart + t^a.letEnd$

Task schedule:

- $hp = LCM\{t^a.period | t^a \in T\}$: The duration of the hyper period

Definitions

Signal:

- $s \in S$: A signal
- $s = \langle style \in \{local, global\} \rangle$: Programming style of the writers
- $W(s) \in T$: Returns the set of tasks that write to s
- $R(s) \subseteq T$: Returns the set of tasks that read from s

Signal writer instances:

- $allWI_s = \left\{ t_i^w \middle| 0 \leq i \in \mathbb{N} < \frac{hp}{t^w.period}, t^w \in W(s) \right\}$: All the possible writer instances for signal s
- $selectedWI_s \subseteq allWI_s$: Writer instances, selected by the designer, that are allowed to write to signal s. All instances must have unique LET end times: $\forall (t_i^a, t_j^b) \in selectedWI_s: t_i^a.letEnd \neq t_j^b.letEnd$
- $selectedWI_s$ is only needed when multiple writer instances have coinciding LET end times. Otherwise, $selectedWI_s$ is equal to $allWI_s$

Definitions

Data-age constraint:

- $t^w \xrightarrow{s,\, \delta_{max}} t^r \in DataAges$: The value of signal s, that is read by task t^r, must not have been written more than δ_{max} time units ago by task t^w
- If a reader does not have a data-age constraint on a signal, then the default behaviour is to read the freshest value, modelled with $\delta_{max} = t^w.period$

Signal usage:

- $uses_s \in Uses$: The usages of signal s by the tasks
- $uses_s = \{\langle t_i^a, \sigma, \phi \rangle\}$: A set of tuples each indicating a change in the usage of signal s by task instance t_i^a at timestamp σ. A usage ϕ can be the start of signal reading ($\mathcal{R}^{start}$) or writing ($\mathcal{W}^{start}$), or the end of signal reading ($\mathcal{R}^{end}$) or writing ($\mathcal{W}^{end}$)
- $GetUses(uses_s, \sigma_1, \sigma_2, \phi)$: Returns from $uses_s$ the usages equal to ϕ with timestamp σ such that $\sigma_1 < \sigma < \sigma_2$
- $GetEarliestUses(uses_s)$: Returns the set of usages with the earliest timestamp

Definitions

Buffer memory:

- $buff_s \in Buffs$: A buffer memory for signal s
- $buff_s = \{e = \{t_i^a\}\}$: A set of buffer elements e that each tracks its set of readers

Buffering schedule:

- $buffSch_s \in BuffSchs$: A buffering schedule for signal s
- $buffSch_s = \{(t_i^a, e)\}$: A set of buffering actions, each defined as a task instance t_i^a and its buffer element e

Buffer lifetime:

- $lifetime_s \in Lifetimes$: A buffer lifetime for signal s
- $lifetime_s = \{interval = (start, end)\}$: A set of intervals, defined as a pair of start and end times

Disjoint buffers:

- $disjBuffs = \{s_1, s_2\} \in DisjBuffs$: An unordered pair of signals with disjoint buffer lifetimes

Design Level: Algorithm and Heuristics

$(\boldsymbol{BuffSchs}, \boldsymbol{AllDisjBuffs}) = \boldsymbol{DesignOptimisations}(\boldsymbol{S}, \boldsymbol{hp}, \boldsymbol{DataAges}, \boldsymbol{AllWI}, \boldsymbol{SelectedWI})$:

$BuffSchs := \emptyset$ // Initialise empty buffering schedules for all signals

$Lifetimes := \emptyset$ // Initialise empty lifetimes for all signal buffers

For each $s \in S$:

// All reader instances will read from signal s

$allRI_s := \left\{ t_i^r \middle| 0 \leq i \in \mathbb{N} < \frac{hp}{t^r.period}, t^r \in R(s) \right\}$

$use_s := \left\{ \langle t_i^r, t_i^r.letStart, \mathcal{R}^{start} \rangle, \langle t_i^r, t_i^r.letEnd, \mathcal{R}^{end} \rangle \middle| t_i^r \in allRI_s \right\}$

If $s.style = global$:

// All writer instances will write to signal s

$use_s := use_s \cup \left\{ \langle t_i^w, t_i^w.letStart, \mathcal{W}^{start} \rangle, \langle t_i^w, t_i^w.letEnd, \mathcal{W}^{end} \rangle \middle| t_i^w \in allWI_s \right\}$

else:

// Not all writer instances may need to write to signal s

$use_s := use_s \cup GetWriterUses(s, selectedWI_s, allRI_s, DataAges)$

$selectedWI_s := \{ t_i^a | \langle t_i^a, \sigma, \delta \rangle \in use_s \}$

end if

$(buffSch_s, lifetime_s) := SBP(use_s, selectedWI_s, hp)$

$BuffSchs := BuffSchs \cup \{buffSch_s\}$

$Lifetimes := Lifetimes \cup \{lifetime_s\}$

end for

$AllDisjBuffs := GetDisjBuffs(Lifetimes)$

return $(BuffSchs, AllDisjBuffs)$

Design Level: Algorithm and Heuristics

$\boldsymbol{uses_s = GetWriterUses(s, selectedWI_s, allRI_s, \delta_{max})}$**:**

$potentialTable_s := \emptyset$ // Start with an empty table of potential reads for each writer instance

$latestTable_s := \emptyset$ // Start with an empty table of latest reads for each writer instance

// Construct the table of potential reads for each writer instance (including the initial value)

For each $t_i^r \in allRI_s$:

  For each $t_j^w \in selectedWI_s \cup \{t_{-1}^w = \langle 0, 0, 0 \rangle\}$:

    If $0 \leq Max(t_i^r.letStart - t_j^w.letEnd) \leq \delta_{max}$ Data-age-aware

      $potentialTable_s\,(t_j^w) := potentialTable_s\,(t_j^w) \cup \{t_i^r\}$

    end if

  end for

end for

Sort $potentialTable_s$ such that $t_j^w.letEnd$ of row n is earlier than or equal to $t_{j'}^{w'}.letEnd$ of row $n+1$

// Determine the latest possible writer instance that each reader instance can read from

For each $potentialTable_s\,(t_j^w)$ and $potentialTable_s(t_{j+1}^w)$, where t_{j+1}^w is the row directly after t_j^w:

  $latestTable_s(t_j^w) := potentialTable_s\,(t_j^w) \setminus potentialTable_s\,(t_{j+1}^w)$

end for

For each $potentialTable_s\,(t_j^w)$, where t_j^w is the last row of $potentialTable_s$:

  $latestTable_s(t_j^w) := potentialTable_s\,(t_j^w)$

end for

Sort $latestTable_s$ such that $t_j^w.letEnd$ of row n is earlier than or equal to $t_{j'}^{w'}.letEnd$ of row $n+1$

... continued ⟶

Design Level: Algorithm and Heuristics

← continued ...

$uses_s := \emptyset$ // Start with an empty set of write usages for signal s
$buffRI_s := \emptyset$ // Start with an empty set of reader instances for which we have found at least one corresponding writer instance

// Record the writer instances that the readers need
For each $latestTable_s(t_j^w)$ in sorted order:
  If $latestTable_s(t_j^w) \notin buffRI_s$
    If $j \neq -1$: // The buffer is always initialised with an initial value
      $uses_s := uses_s \cup \{\langle t_j^w, t_j^w.letStart, \mathcal{W}^{start}\rangle, \langle t_j^w, t_j^w.letEnd, \mathcal{W}^{end}\rangle\}$
    end if
    $buffRI_s := buffRI_s \cup potentialTable_s(t_j^w)$
  end if
end for

// Last writer instance will write the initial value for the next hyper period
$uses_s := uses_s \cup \{\langle t_j^w, t_j^w.letStart, \mathcal{W}^{start}\rangle, \langle t_j^w, t_j^w.letEnd, \mathcal{W}^{end}\rangle\}$ where t_j^w is from the last row of $potentialTable_s$

return $uses_s$

Design Level: Algorithm and Heuristics

$(\boldsymbol{buffSch_s, lifetime_s}) = \boldsymbol{SBP(uses_s, selectedWI_s, hp)}$:

$buffSch_s := \emptyset$ // Initialise an empty buffering schedule for signal s

$buff_s := \{e_o := \emptyset\}$ // Initialise a buffer for signal s, where element e_0 holds an initial value

$lifetime_s := \emptyset$ // Start with an empty set of lifetime intervals

$interval := (start := 0, end := 0)$ // Start with an initial lifetime interval

$current := (\&prev := \&e_0, inst := null)$ // Tracks the current prev to read from

$Prev := \{\&prev^{t^a} := \&e_0 | t^a \in W(s)\}$ // References to the writers' prev buffer elements

$Next := \{\&next^{t^a} := null | t^a \in W(s)\}$ // References to the writers' next buffer elements

Repeat until $uses_s = \emptyset$:

// Get all the signal usages with the earliest timestamp

$earliestUses_s := GetEarliestUses(uses_s)$

$uses_s := uses_s \setminus \{earliestUses_s\}$

// At the end of a reader's LET, it no longer needs its buffer element

$readersEnd := \{use.t_i^a | use.\phi = \mathcal{R}^{end}, use \in earliestUses_s\}$

$buff_s := \{e \setminus \{readersEnd\} | e \in buff_s\}$

// At the end of a writer's LET, its new value becomes available

$writersEnd := \{use.t_i^a | use.\phi = \mathcal{W}^{end}, use \in earliestUses_s\}$

$\forall t_i^a \in writersEnd: \&prev^{t^a} := \&next^{t^a}; \&next^{t^a} := null$

If $\exists t_i^a \in writersEnd \cap selectedWriterInstances_s$, then $current := (\&prev^{t^a}, t_i^a)$ end if

... continued ⟶

Design Level: Algorithm and Heuristics

← continued ...

// At the start of a reader's LET, it occupies the writer's previous buffer element
$readersStart := \{use.t_i^a | use.\phi = \mathcal{R}^{start}, use \in earliestUses_s\}$
$current.prev := current.prev \cup readersStart$
$buffSch_s := buffSch_s \cup \{(t_i^a, current.prev) | t_i^a \in readersStart\}$

If $\exists t_i^a \in readersStart$:
  $maxLetEnd := \max\{t_i^a.letEnd | t_i^a \in readersStart\}$
  If $current.inst = null$: // Initial value is needed, because no writer has finished
    $interval := (0, maxLetEnd)$
  else if $current.inst.letStart \leq interval.end$: // Writer instance overlaps with the current interval
    $interval.end := \max(interval.end, maxLetEnd)$
  else if $interval = (0,0)$: // Initial value is never used
    $interval := (current.inst.letStart, maxLetEnd)$
  else: // Writer instance is disjoint from the current interval
    $lifetime_s := lifetime_s \cup \{interval\}$
    $interval := (current.inst.letStart, maxLetEnd)$
  end if
end if

... continued →

Design Level: Algorithm and Heuristics

← continued ...

// At the start of a writer's LET, it finds a free buffer element to use

$writersStart := \{use.t_i^a | use.\phi = \mathcal{W}^{start}, use \in earliestUses_s\}$

$occupiedBuff_s := Next \cup \{e \in buff_s | e \neq \emptyset\}$
$\cup \{current.prev | GetUses(uses_s, current.inst.letStart, current.inst.letEnd, r^{start}) \neq \emptyset\}$

For each $t_i^a \in writersStart$, with $prev^{t^a} \in Prev$ and $next^{t^a} \in Next$:

If $prev^{t^a} \notin occupiedBuff_s$:

$\&next^{t^a} := \&prev^{t^a}$

else:

If $\exists e \in buff_s \setminus occupiedBuff_s$, then $\&next^{t^a} := \&e$,

else $buff_s := buff_s \cup \{e := \emptyset\}$; $\&next^{t^a} := \&e$ end if

$buffSch_s := buffSch_s \cup \{(t_i^a, next^{t^a})\}$

end if

$occupiedBuff_s := occupiedBuff_s \cup \{next^{t^a}\}$

$lifetime_s := lifetime_s \cup \{(t_i^a.letStart, t_i^a.letEnd)\}$ // All writer instances contribute to the lifetime

end for

end repeat

// Last writer instance will write the initial value for the next hyper period

$lifetime_s := lifetime_s \cup \{interval, (current.inst.letStart, hp)\}$

If $current.prev \neq \&e_0$, then copy the buffered value at $current.prev$ to e_0 at the end of the hyper period

Return $(Reduce(buffSch_s), lifetime_s)$

Design Level: Algorithm and Heuristics

$\boldsymbol{buffSch_s = Reduce(buffSch_s)}$**:**

// When a reader has two consecutive buffering actions to the same buffer element,
// the second action is redundant and can be eliminated

For each $\{(t_i^a, e_x), (t_k^a, e_x)\} \subseteq buffSch_s$, $t \in R(s)$, and $\nexists(t_j^a, e_y) \in buffSch_s$ where $i < j < k$:

$buffSch_s := buffSch_s \setminus \{(t_k^a, e_x)\}$

return $buffSch_s$

$\boldsymbol{DisjBuffs = GetDisjBuffs(Lifetimes)}$**:**

$DisjBuffs := \emptyset$ // Initialise an empty set of disjoint buffers

For each $\{lifetime_{s1}, lifetime_{s2}\} \subseteq Lifetimes$:

// The lifetimes of s1 and s2 are disjoint if none of their intervals intersect. This algorithm is unoptimised for
// the purpose of understandability. Optimised interval calculations in literature, e.g., A. Gavryushkin, B.
// Khoussainov, M. Kokho, and J. Liu. *Dynamic Algorithms for Multimachine Interval Scheduling Through*
// *Analysis of Idle Intervals*. Algorithmica, Volume 76, Issue 4, pp 1160–1180, December 2016

If $\nexists interval_{s1} \in lifetime_{s1}$ and $\nexists interval_{s2} \in lifetime_{s2}$,
where $(interval_{s1}.start \leq interval_{s2}.start \leq interval_{s1}.end)$
or $(interval_{s2}.start \leq interval_{s1}.start \leq interval_{s2}.end)$:

$DisjBuffs := DisjBuffs \cup \{\{s1, s2\}\}$

end if

end for

return $DisjBuffs$

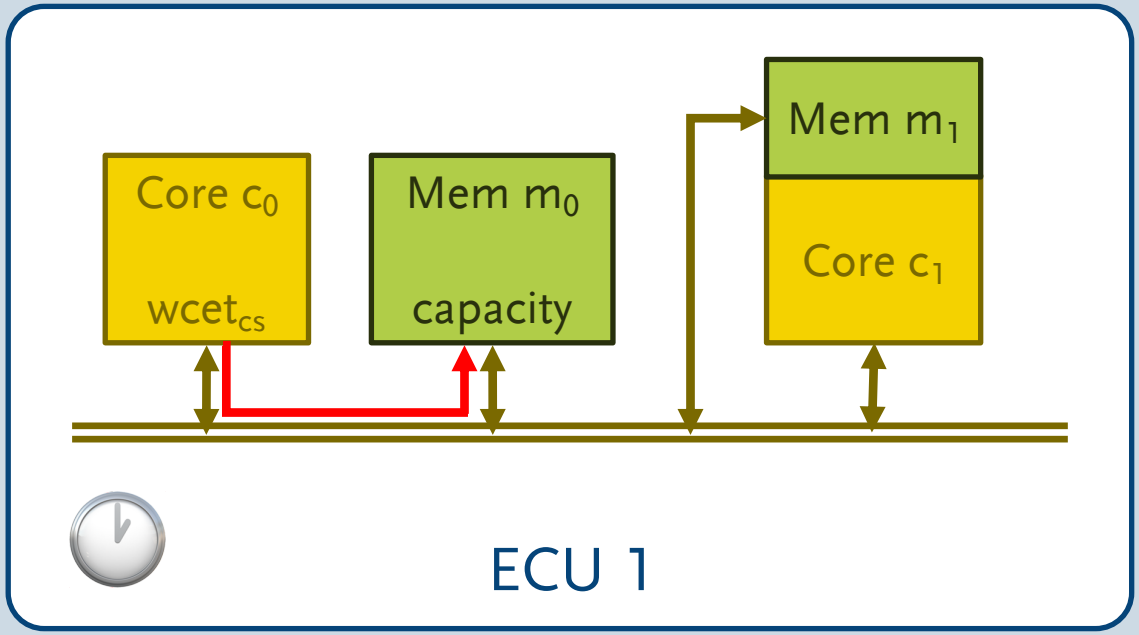
Informal Description: Deployment Level (MILP)

Deployment Level Optimisations

Model the system with linear constraints on real, integer, and Boolean variables (mixed-integer linear programming, MILP)

- Simultaneously:
  - Allocate signal buffers to memory modules
  - Allocate tasks to cores
  - Select a buffering protocol for each signal
  - Check the task schedulability of each core
    by calculating the latencies for tasks to access each signal buffer, and
    by calculating each task's worst-case execution time
- Minimise:
  - Total time allocated in the hyper period schedule for task execution
- Limitations:
  - Does not explore possible buffer merges, to restrict the search space
  - Does not break MILP symmetry: Avoid solving equivalent solutions, e.g., due to equivalent cores, memory modules, or LET tasks

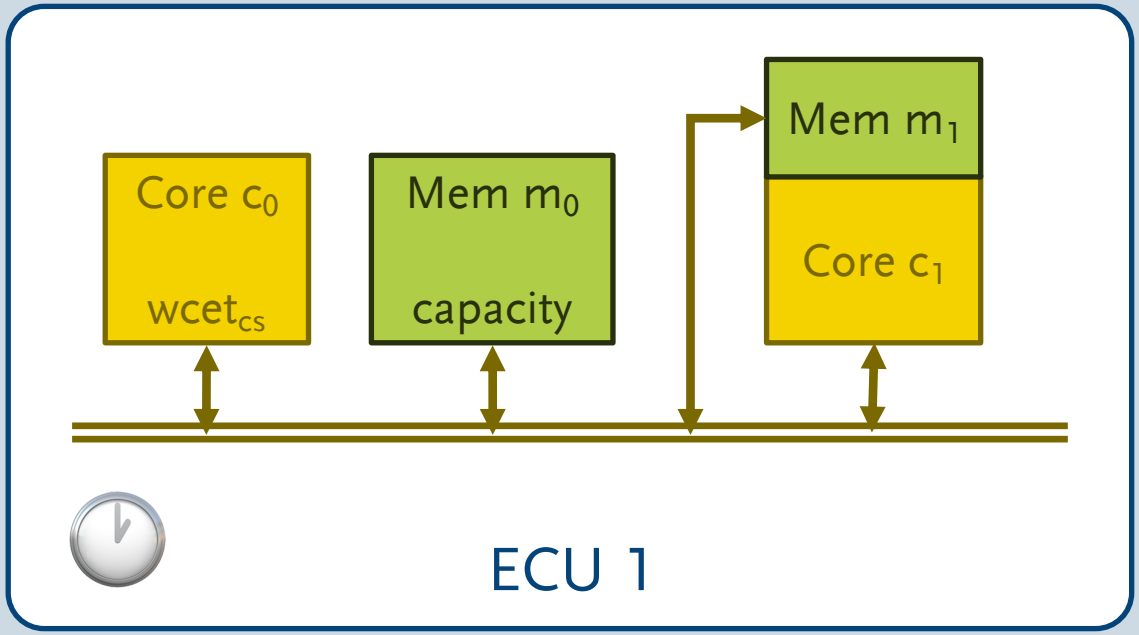
Intuition: MILP



Core-to-memory latency = Bus latency + Memory latency

- Time for a core to write to or read from a memory location
- Bus latency: Arbitration policy and number of buses to traverse
- Under worst-case assumptions: Very pessimistic
- All data widths are equal

Intuition: MILP



WCET	Core c_0	Core c_1
sbp	20 ns	40 ns
ptp	10 ns	20 ns
cs	40 ns	70 ns

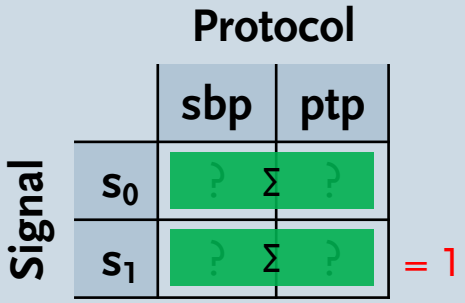
Memory	Size
m_0	1024 MiB
m_1	256 KiB

Path = Memory	Core c_0	Core c_1
m_0	20 ns	10 ns
m_1	15 ns	15 ns

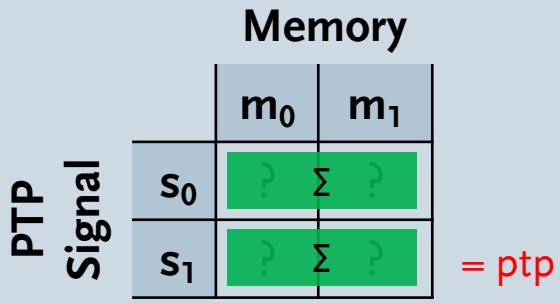
Signal	Trans.	Size	n
s_0	2	32 KiB	3
s_1	1	16 KiB	4

Intuition: MILP

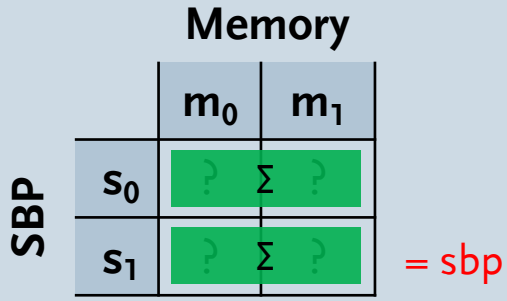
Selection of buffering protocols:

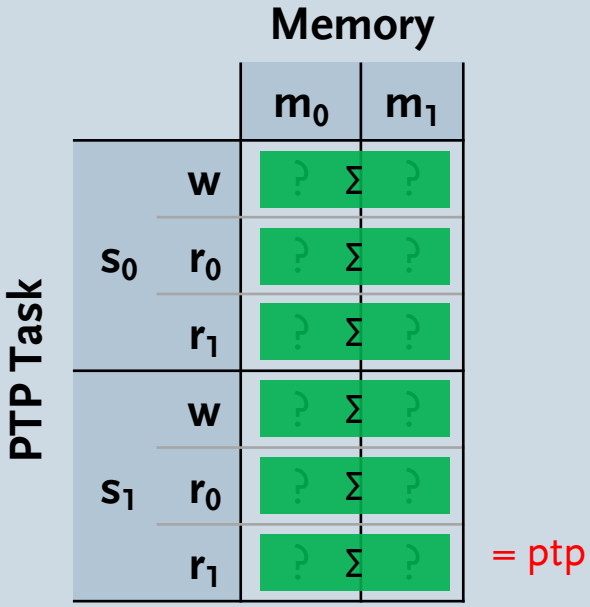


PTP buffer-to-memory allocations:



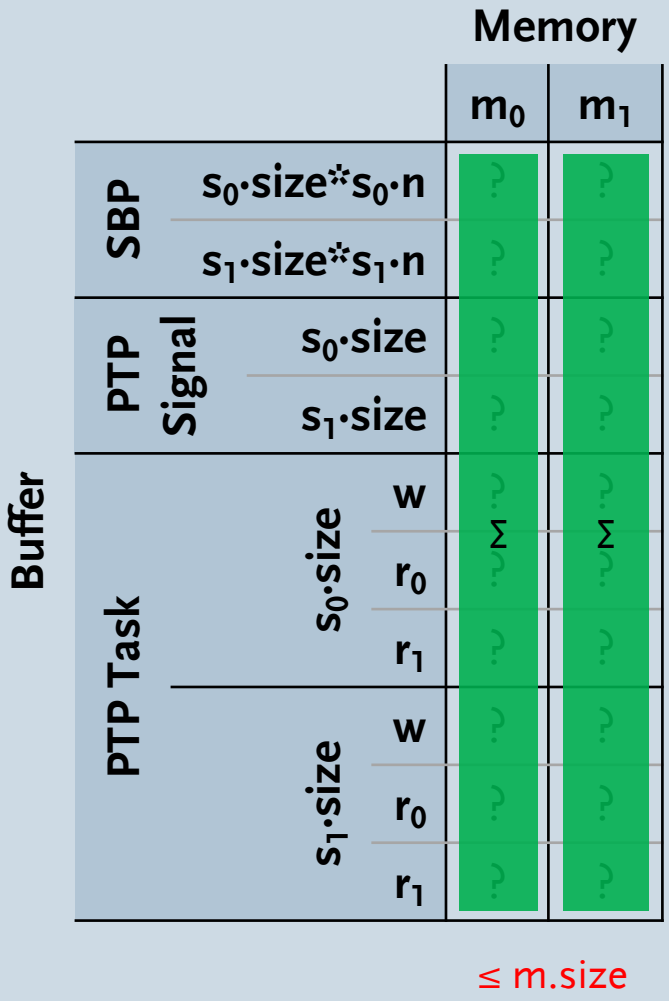
SBP buffer-to-memory allocations:



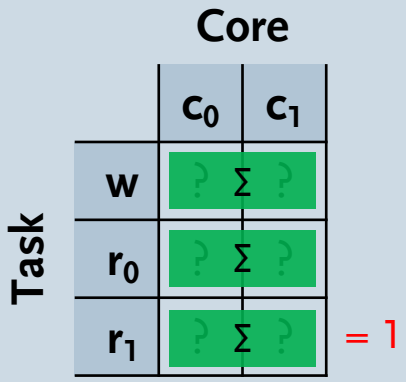


Intuition: MILP

Memory utilisations:



Task-to-core allocations:



Buffers and tasks can be allocated to specific memory modules and cores by fixing their variables to constants

Intuition: MILP

Task-to-signal buffer latencies:

Memory \ Core	c_0	c_1
m_0	…	…
m_1	…	…

×

Buffer \ Memory	m_0	m_1
sbp	?	?
ptp	?	?

×

Task \ Core	c_0	c_1
w	?	?
r_0	?	?
r_1	?	?

×

Signal	Trans.	Size	n
s_0	…	…	…
s_1	…	…	…

Result:

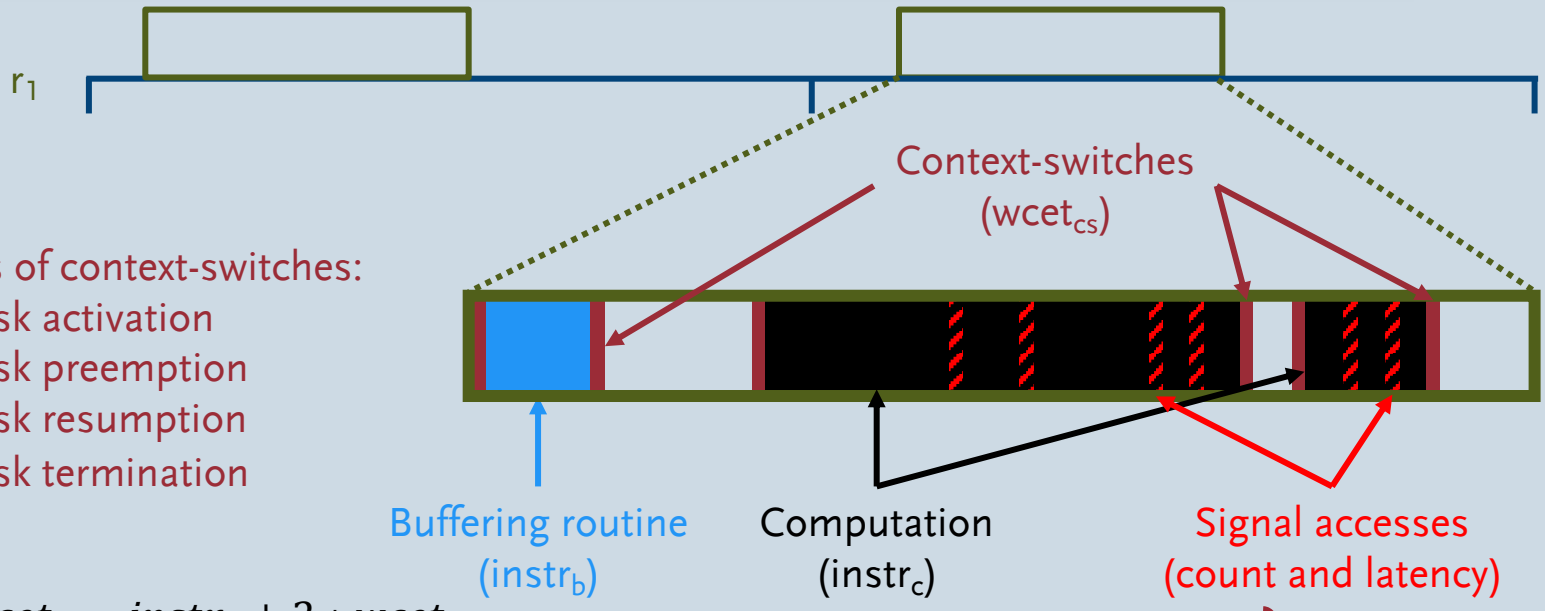
Task-to-Core

Buffer-to-Memory		w		r_0		r_1	
		c_0	c_1	c_0	c_1	c_0	c_1
PTP,SBP	m_0	? ? Σ ? ?		? ? Σ ? ?		? ? Σ ? ?	
	m_1						
PTP,SBP	m_0	? ? Σ ? ?		? ? Σ ? ?		? ? Σ ? ?	
	m_1						

Intuition: MILP

Can be improved with more elaborate timing analysis techniques

R. Wilhelm et al., *The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools*. ACM TECS. 2008



Types of context-switches:

- Task activation
- Task preemption
- Task resumption
- Task termination

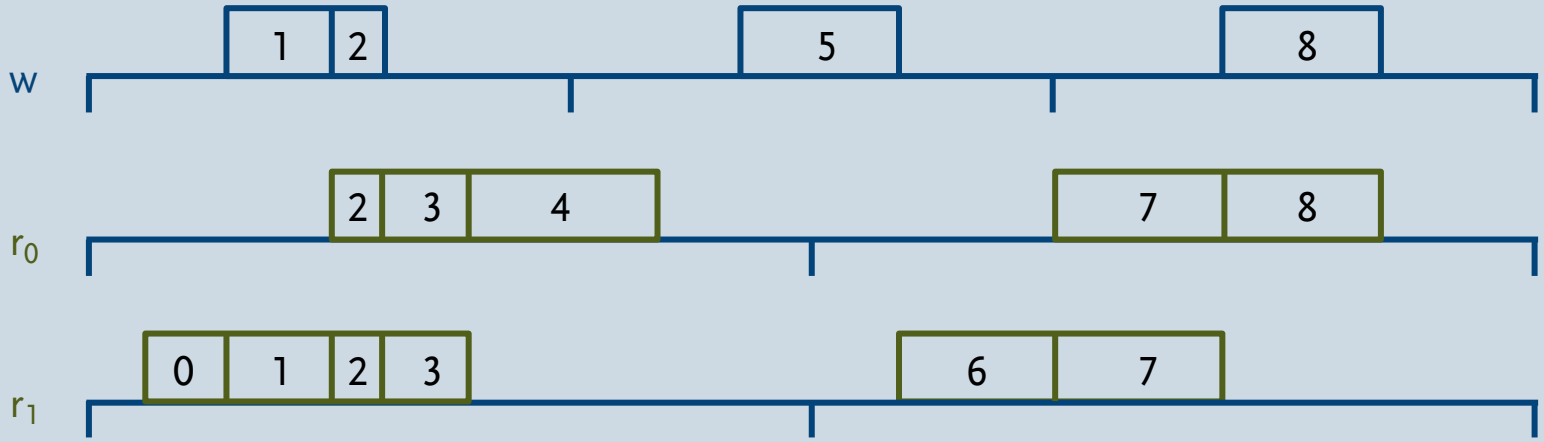
$$r_1.wcet_b = instr_b + 2 * wcet_{cs}$$

$$r_1.wcet_c = instr_c + wcet_{cs} * r_1.cs + \sum_{(s,n) \in r_1.acc} (latency.r_1.s * s.w * n)$$

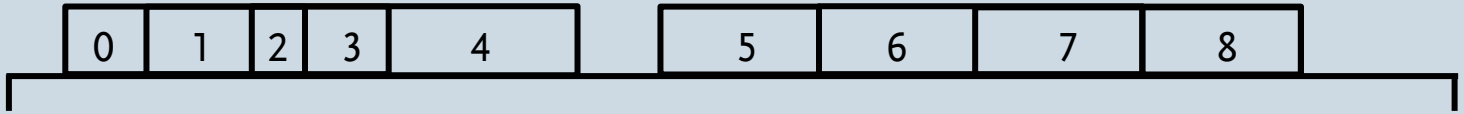
$$r_1.wcet_b + r_1.wcet_c \leq r_1.letEnd - r_1.letStart$$

Over all the core-specific WCETs

Intuition: MILP



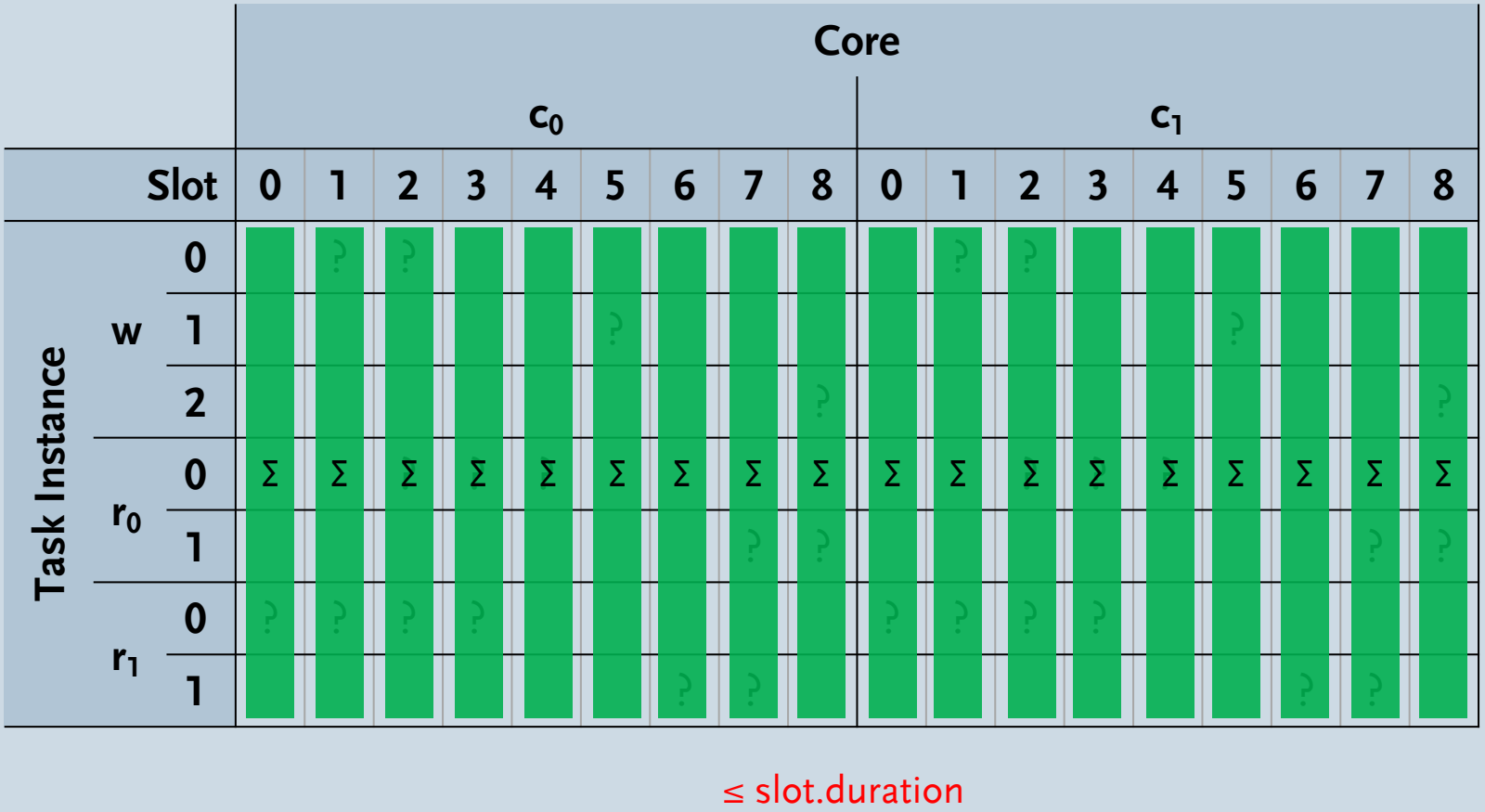
Hyper period slots:



	Task														
	w				r_0					r_1					
Instance	0		1	2	0			1		0				1	
Slot	1	2	5	8	2	3	4	7	8	0	1	2	3	6	7
Allocation	?	Σ	?	Σ	Σ	?	Σ	?	Σ	?	?	Σ	?	?	?	Σ	?

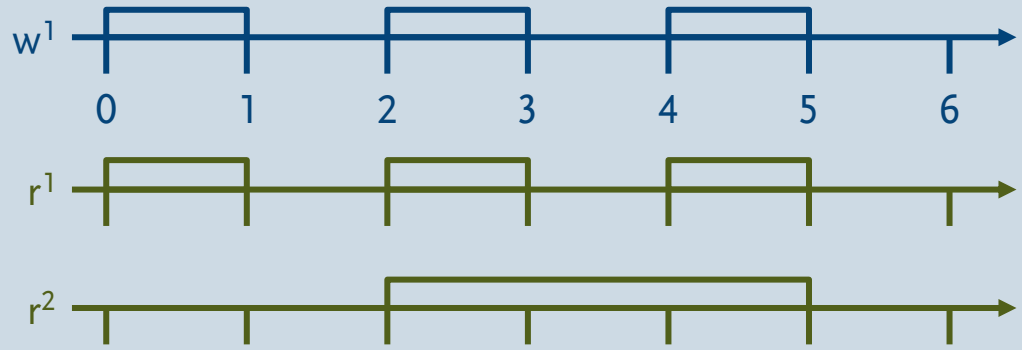
$\geq t.wcet_s + t.wcet_c$

Intuition: MILP



Deployment Level: Scheduling Hint

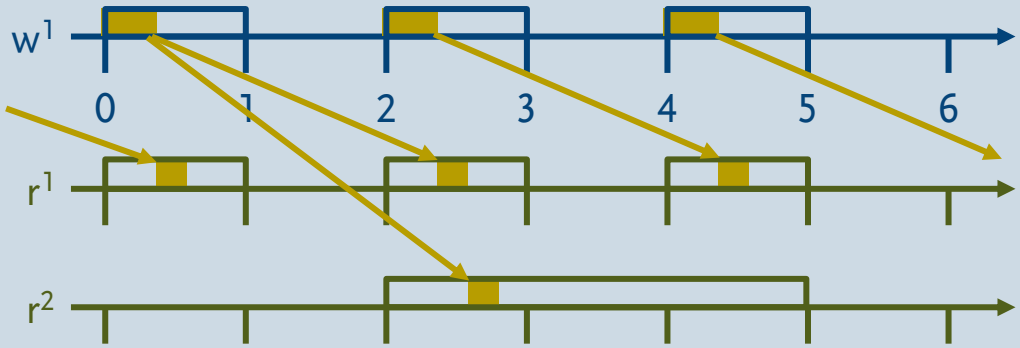
Design level: Example assuming all writer instances need to write to the signal



3 buffer elements needed during time units 4—5

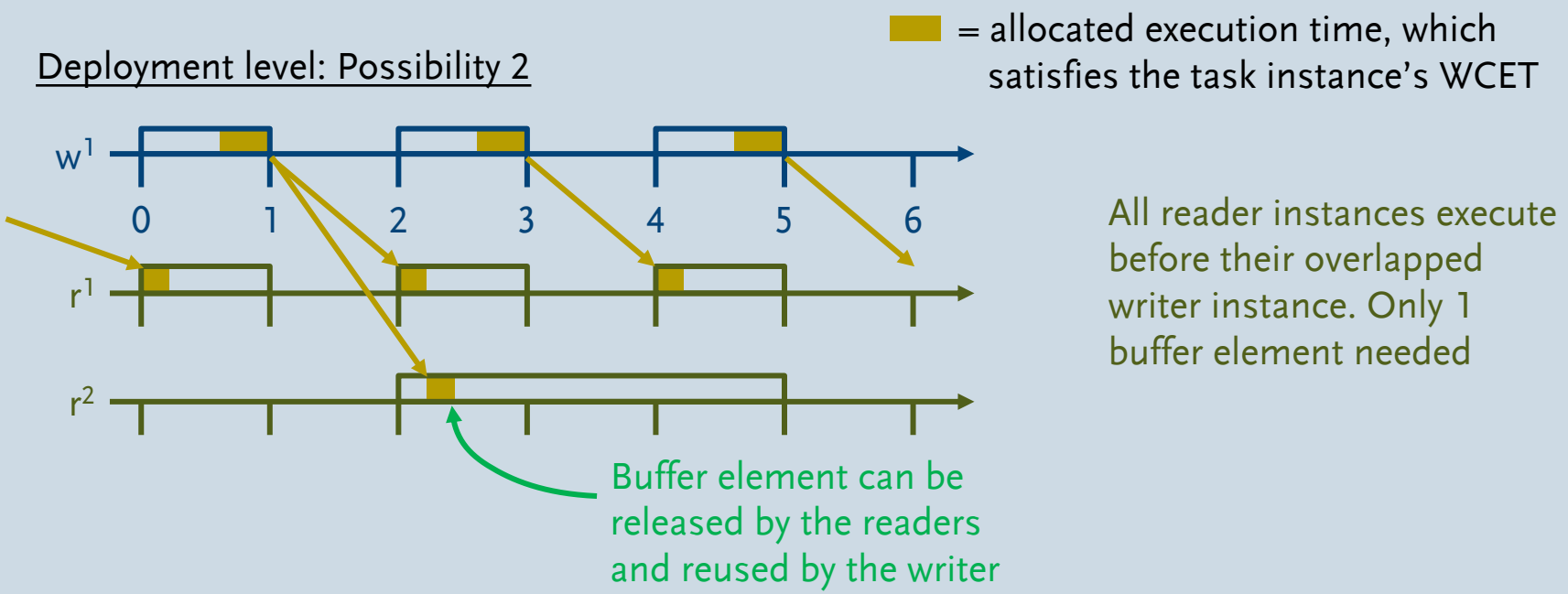
= allocated execution time, which satisfies the task instance's WCET

Deployment level: Possibility 1



Writer instances execute before their overlapped reader instance.
2 buffer elements needed during time units 0—1, 2—3, and 4—5

Deployment Level: Scheduling Hint



A signal's buffer memory could be reduced by allocating execution time as early as possible for its readers, and as late as possible for its writers

- Tasks with signal reads **and** writes can only be optimised for one or the other
- Unlike PTP, which requires buffer updates at LET boundaries, SBP buffer indexes can be updated at the start of a task's computation

Deployment Level: Scheduling Hint

Re-run DesignOptimisations(...) for the signals selected to use SBP

- Interested in the earliest time that SBP buffers can be released, and the latest time that they must be acquired
- Extend the definition of a task with $t_i^a.execStart$ and $t_i^a.execEnd$ to denote the start and end of task t_i^a's allocated execution time

- When a reader completes its computations, release its buffer element: $use_s := use_s \cup \{\langle t_i^r, t_i^r.letStart, \mathcal{R}^{start}\rangle, \langle t_i^r, t_i^r.execEnd, \mathcal{R}^{end}\rangle\}$
- When a writer begins its computations, acquire a free buffer element: $use_s := use_s \cup \{\langle t_i^w, t_i^w.execStart, \mathcal{W}^{start}\rangle, \langle t_i^w, t_i^w.letEnd, \mathcal{W}^{end}\rangle\}$

- Optimised SBP buffering schedules and buffer sizes
- Pairs of SBP buffers with disjoint lifetimes

Deployment Level: Merging SBP Buffers

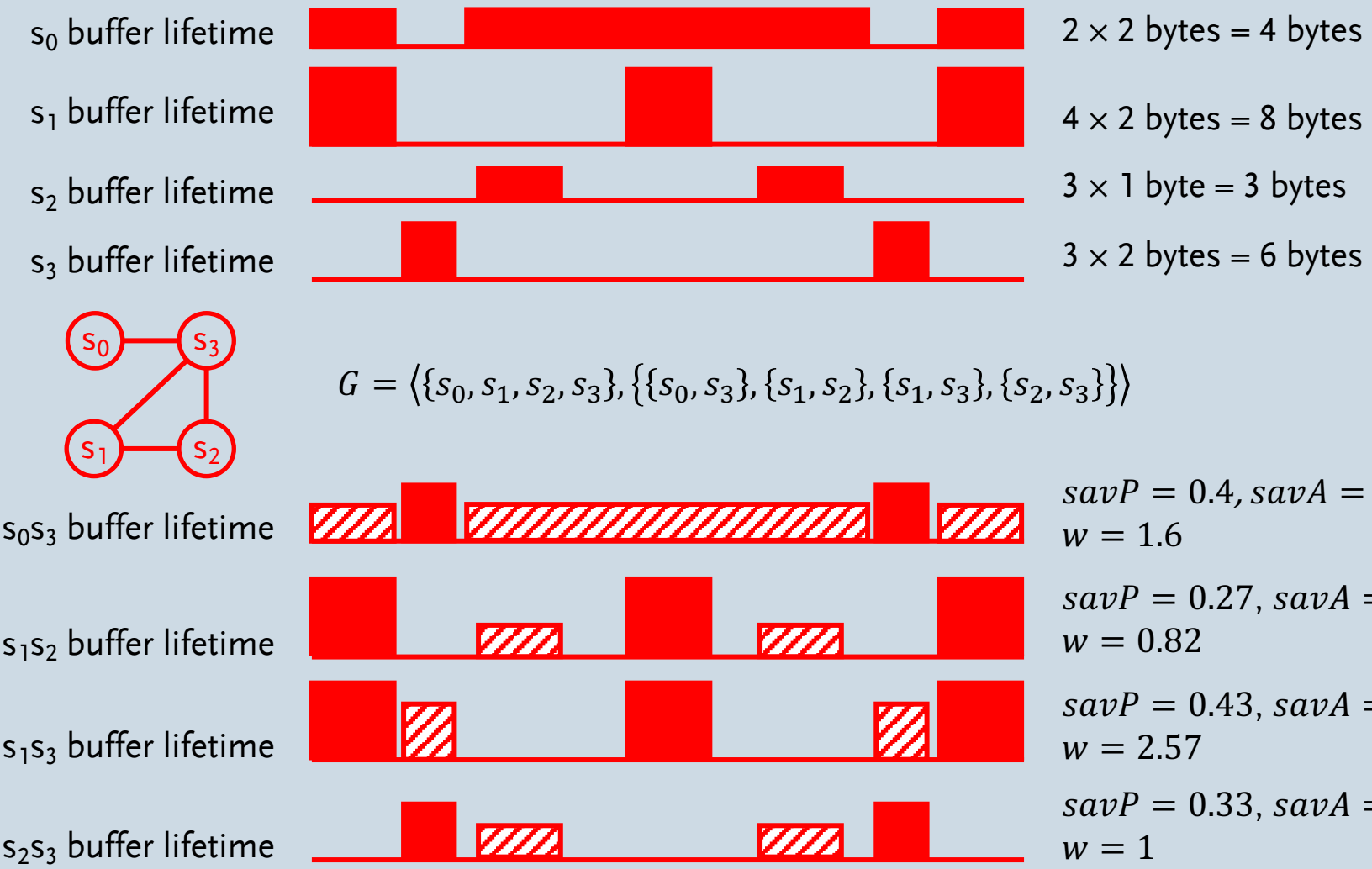
For each memory module, merge the disjoint SBP signal buffers

- Graph of disjoint SBP signal buffers: $G = \langle S, E \rangle$
- A vertex $s \in S$ is a signal buffer, and two vertices with disjoint lifetimes are connected by an undirected edge $e \in E$
- Each clique $C \subseteq G$ is a group of signal buffers that can share the same block of memory
- Cliques could overlap
- Selection of clique C is weighted by proportionate ($savP_C$) and absolute ($savA_C$) memory savings: $w_c = savP_C * savA_C$

$$savP_C = 1 - \frac{\max_{s \in C.S}(s.size * s.n)}{\sum_{s \in C.S}(s.size * s.n)}$$

$$savA_C = \sum_{s \in C.S}(s.size * s.n) - \max_{s \in C.S}(s.size * s.n)$$

Deployment Level: Merging SBP Buffers



$MergedBuffs = \{\{s_1, s_3\}\}$, 15 bytes in total (savings of 6 bytes)

Deployment Level: Merging SBP Buffers

A merge with more savings:

s_0s_3 buffer lifetime

$savP = 0.4, savA = 4$ b
$w = 1.6$

s_1s_2 buffer lifetime

$savP = 0.27, savA = 3$ b
$w = 0.82$

$MergedBuffs = \{\{s_0, s_3\}, \{s_1, s_2\}\}$, 14 bytes in total (savings of 7 bytes)

A merge with the most savings:

s_0 buffer lifetime

2×2 bytes = 4 bytes

$s_1s_2s_3$ buffer lifetime

$savP = 0.53, savA = 9$ b
$w = 4.76$

$MergedBuffs = \{\{s_0\}, \{s_1, s_2, s_3\}\}$, 12 bytes in total (savings of 9 bytes)

Formal Description: Deployment Level (MILP)

Additional Definitions

Heterogeneous core:

- $c \in C$: An available core c
- $wcet_{sbp}.c$: Worst-case (physical) time for updating an SBP buffer index on core c
- $wcet_{ptp}.c$: Worst-case (physical) time for core c to prepare the copying a signal
- $wcet_{cs}.c$: Worst-case (physical) time for context-switching on core c

Memory module:

- $m = \langle size \rangle \in M$: An available memory module with a $size$ in bytes. All modules have same data width

Core-to-memory access latency:

- $path = \langle c, m, l \rangle \in Paths$: An access from core c to memory module m has a latency of l. The latency is statically computed to consider the worst-case situation, which depends on the buses that need to be traversed and their arbitration policies, and the latencies of the memory modules

Additional Definitions

Signal:

- $s = \langle size, n, w \rangle \in S$: A signal has a data $size$ in bytes, the number n of required buffer elements, and the number w of bus transfers for each access

Hyper period slot:

- $slot = \langle d \rangle \in Slots$: The LETs in the hyper period schedule are projected onto a single timeline and split into uniquely identifiable slots, demarcated by the LET boundaries. Each slot has a duration d

Additional Definitions

LET task:

- $t = \langle period, letStart, letEnd, acc, instr, cs \rangle \in T$:
  where acc gives the maximum number of times that a signal s is accessed, $instr$ gives the WCETs of the instructions on each core c, and cs is the maximum number of context-switches that could occur
- $acc = \{(s, n)\}$: The maximum number of times that signal s is accessed is n
- $instr = \{(c, wcet)\}$: The $wcet$ (physical time) of the instructions on core c

LET task instance:

- $t_i = \langle periodStart, letStart, letEnd, slots = \{slot\} \rangle$: where $slots$ is t_i's LET as a set of corresponding hyper period slots

Deployment Level: MILP

Buffers

The Boolean variables $s.sbp$ and $s.ptp$ indicate whether SBP ($s.sbp = 1$) or PTP ($s.ptp = 1$), respectively, is used for signal s:

$$\forall s \in S\colon\ s.sbp + s.ptp = 1$$

SBP:

If SBP is used, the Boolean variable $sbp_s.m$ indicates whether the SBP buffer for signal s is allocated to memory module m ($sbp_s.m = 1$). The buffer can only be allocated to one memory module:

$$\forall s \in S\colon \sum_{m \in M} sbp_s.m = s.sbp$$

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

$$\forall m \in M\colon\ m.sbp = \sum_{s \in S} (sbp_s.m * s.size * s.n)$$

Deployment Level: MILP

PTP:

If PTP is used, the Boolean variable $ptp_s.t.m$ indicates whether task t's PTP buffer for signal s is allocated to memory module m ($ptp_s.t.m = 1$). Moreover, the Boolean variable $ptp_s.m$ indicates whether the signal s itself is allocated to memory module m ($ptp_s.m = 1$). Each buffer can only be allocated to one memory module:

$$\forall t \in T, \forall (s,n) \in t.acc: \sum_{m \in M} ptp_s.t.m = s.ptp$$

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

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

$$\forall m \in M: m.ptp = \sum_{t \in T} \sum_{(s,n) \in t.acc} (ptp_s.t.m * s.size) + \sum_{s \in S} (ptp_s.m * s.size)$$

Finally, the total utilisation of a memory module m due to SBP and PTP is limited by its size:

$$\forall m \in M: m.ptp + m.sbp \leq m.size$$

Deployment Level: MILP

Tasks

The Boolean variable $t.c$ indicates whether task t is allocated to core c ($t.c = 1$). A task can only be allocated to one core:

$$\forall t \in T: \sum_{c \in C} t.c = 1$$

We use the following Boolean variables to express relationships between task, core, buffer, and memory allocations:

- $t.c.s.sbp = 1$ only when task t is on core c and signal s uses SBP,
- $t.c.sbp_s.m = 1$ only when task t is on core c and signal s' SBP buffer is on memory m,
- $t.c.ptp_s.t.m = 1$ only when task t is on core c and its PTP buffer for s is on memory m, and
- $t.c.ptp_s.m = 1$ only when task t is on core c and signal s' PTP buffer is on memory m.

$$0 \leq t.c + s.sbp - 2 * t.c.s.sbp \leq 1$$
$$0 \leq t.c + sbp_s.m - 2 * t.c.sbp_s.m \leq 1$$
$$0 \leq t.c + ptp_s.t.m - 2 * t.c.ptp_s.t.m \leq 1$$
$$0 \leq t.c + ptp_s.m - 2 * t.c.ptp_s.m \leq 1$$

Deployment Level: MILP

Worst-case buffer management time:

The worst-case time to manage task t's buffers is tracked by the (positive) real variable $t.wcet_b$. It is the sum of the context-switching time ($wcet_{cs}.c$), and the worst-case time to initialise the SBP buffer indexes ($wcet_{sbp}.c$) and to copy the signals to and from the PTP buffers:

$$\forall t \in T:\ t.wcet_b = 2 * \sum_{c \in C} (t.c * wcet_{cs}.c) + \sum_{c \in C} \sum_{(s,n) \in t.acc} \left(t.c.s.sbp * wcet_{sbp}.c\right)$$
$$+ \sum_{c \in C} \sum_{m \in M} \sum_{(s,n) \in t.acc} \left(t.c.ptp_s.t.m * \left(wcet_{ptp}.c + path.c.m * s.w\right)\right)$$
$$+ \sum_{c \in C} \sum_{m \in M} \sum_{(s,n) \in t.acc} \left(t.c.ptp_s.m * path.c.m * s.w\right)$$

Deployment Level: MILP

Worst-case computation time:

The worst-case time to execute task t's computations is tracked by the (positive) real variable $t.wcet_c$. It is the sum of the worst-case time to execute the instructions ($t.instr.c.wcet$), the context-switching time ($wcet_{cs}.c$), and the time to perform all buffered signal accesses:

$$\forall t \in T:\ t.wcet_c = \sum_{c \in C} (t.c * t.instr.c.wcet) + \sum_{c \in C} (t.c * wcet_{cs}.c * t.cs)$$
$$+ \sum_{c \in C} \sum_{m \in M} \sum_{(s,n) \in t.acc} (t.c.sbp_s.m * path.c.m * s.w * n)$$
$$+ \sum_{c \in C} \sum_{m \in M} \sum_{(s,n) \in t.acc} (t.c.ptp_s.t.m * path.c.m * s.w * n)$$

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

$$\forall t \in T: t.wcet_b + t.wcet_c \leq t.letEnd - t.letStart$$

Deployment Level: MILP

Schedulability

The time that can be allocated in a hyper period $slot$ for task t_i's execution is tracked by the (positive) real variable $slot.t_i.alloc$. Enough time must be allocated for the task's WCET:

$$\forall t \in T, \forall t_i: \sum_{slot \in t_i.slots} (slot.t_i.alloc) \geq t.wcet_b + t.wcet_c$$

On each core, the total time allocated to a $slot$ cannot be greater than its duration d:

$$\forall c \in C, \forall slot \in Slots: \sum_{\substack{t \in T, t_i, \\ slot \in t_i.slots,}} (t.c * slot.t_i.alloc) \leq slot.d$$

We replace the product $t.c * slot.t_i.alloc$ with its linearised form, $t_i.c.slot.alloc$:

$$t_i.c.slot.alloc \leq t.c * slot.d$$
$$t_i.c.slot.alloc \leq slot.t_i.alloc$$
$$t_i.c.slot.alloc \geq slot.t_i.alloc - (1 - t.c) * slot.d$$
$$t_i.c.slot.alloc \geq 0$$

Deployment Level: MILP

Objective: Minimise the total time allocated in the hyper period schedule for task execution

$$Minimise: \sum_{t \in T} \sum_{slot \in t_i.slots} (slot.t_i.alloc)$$

Deployment Level: MILP

Summary of notations used in the MILP formulation

Variable Type	Notations
Boolean	$s.sbp, s.ptp, sbp_s.m, ptp_s.m, ptp_s.t.m, t.c,$ $t.c.s.sbp, t.c.sbp_s.m, t.c.ptp_s.m, t.c.ptp_s.t.m$
Integer	$m.sbp, m.ptp$
Real	$t.wcet_b, t.wcet_c, slot.t_i, alloc, t_i.c.slot.alloc$

Constant Type	Notations
Integer	$s.size, s.n, s.w, t.acc.n, m.size, t.cs$
Real	$wcet_{cs}.c, wcet_{sbp}.c, wcet_{ptp}.c, path.c.m,$ $t.instr.c.wcet, t.letStart, t.letEnd, slot.d$

Deployment Level: Scheduling Hint

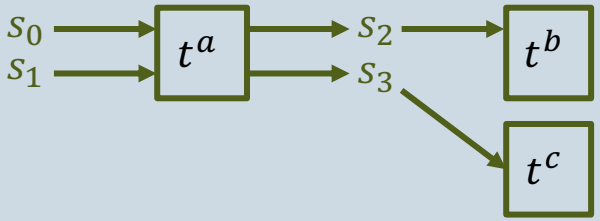
$(\boldsymbol{OptAsWriter}, \boldsymbol{OptAsReaders}) = \boldsymbol{GetSchedulingHints}(\boldsymbol{T})$:

// Each task has an associated set of signals that it writes to and reads from:

// $t^a.writes = \{s\}$ and $t^a.reads = \{s\}$

// Initialise empty sets for the tasks that should be optimised for writing or for reading

$OptAsWriters := \emptyset,\ OptAsReaders := \emptyset$



// Tasks dedicated to writing should be optimised for writing

$OptAsWriters := \{t^a \in T | t^a.reads = \emptyset\}$

// Tasks dedicated to reading, or tasks that write to and read from the same signal, should be

// optimised for reading. The successors of dedicated writers should be optimised for reading.

$$OptAsReaders := \{t^a \in T | t^a.writes = \emptyset \vee t^a.writes \cap t^a.reads \neq \emptyset\} \cup \{t^b \in R(s) | s \in t^a.writes, t^a \in OptAsWriters\}$$

Deployment Level: Scheduling Hint

// Decide which of the remaining tasks should be optimised as writers or as readers.
// Only consider the tasks with successors that have not been selected for optimisation

While $\exists t^a \in T \setminus (OptAsWriters \cup OptAsReaders)$,
where $\{t^b \in R(s) | s \in t^a.writes\} \cap OptAsWriters = \emptyset$ and $|t^a.writes| = Min$:

$OptAsWriters := OptAsWriters \cup \{t^a\}$

$OptAsReaders := OptAsReaders \cup \{t^b \in R(s) | s \in t^a.writes\}$

end for

// All tasks in $OptAsWriters$ should be scheduled for writing
// All tasks in $OptAsReaders$ should be scheduled for reading
// All other tasks can be freely scheduled for writing or for reading

Return $(OptAsWriter, OptAsReaders)$

// Hyper period scheduler prioritises tasks in $OptAsWriter$ by ascending $|t^a.writes|$
// Hyper period scheduler prioritises tasks in $OptAsReaders$ by ascending $|t^a.reads|$

Deployment Level: Merging SBP Buffers

$\boldsymbol{AllMergedBuffs = MergeSBP(AllDisjBuffs, S, MILP)}$:

$AllMergedBuffs := \emptyset$ // Initialise an empty set of merged buffers for all memories

For each $m \in MILP.M$:

// Construct graph of disjoint SBP signal buffers for memory module m

$S_{SBP} := \{s \in S | MILP.s.sbp = 1 \wedge MILP.sbp_s.m = 1\}$

$DisjBuffs := \{disjBuffs \in AllDisjBuffs | disjBuffs \cap S_{SBP} \neq \emptyset\}$

$G := \langle S = S_{SBP}, E = DisjBuffs \rangle$

$MergedBuffs_m := \emptyset$ // Initialise an empty set of merged buffers for memory module m

Repeat until $\nexists C$ where $C := GetWeightedClique(G)$:

$MergedBuffs_m := MergedBuffs_m \cup \{C\}$ // All buffers in clique C will be merged

$G := \langle S = G.S \setminus C.S, E = \{disjBuffs \in G.E | disjBuffs \cap C.S = \emptyset\} \rangle$ // Remove clique C

End repeat

$AllMergedBuffs := AllMergedBuffs \cup \{MergedBuffs_m\}$

End for

return $AllMergedBuffs$

Deployment Level: Merging SBP Buffers

$C = GetWeightedClique(G)$:

$found := \langle C = null, w = 0 \rangle$ // Record the clique with the greatest weight

// Get all the cliques in graph G
// C. Bron and J. Kerbosch. *Algorithm 457: Finding all Cliques of an Undirected Graph.*
// Communications of the ACM, vol. 9, no. 16, pp. 575 – 577, 1973

$C_{All} := GetAllCliques(G)$

For each $C \in C_{All}$:

// Find a clique with the greatest weight

$savP_C := 1 - \frac{\max_{s \in C.S}(s.size * s.n)}{\sum_{s \in C.S}(s.size * s.n)}$

$savA_C := \sum_{s \in C.S}(s.size * s.n) - \max_{s \in C.S}(s.size * s.n)$

$w_{new} := savP_C * savA_C$

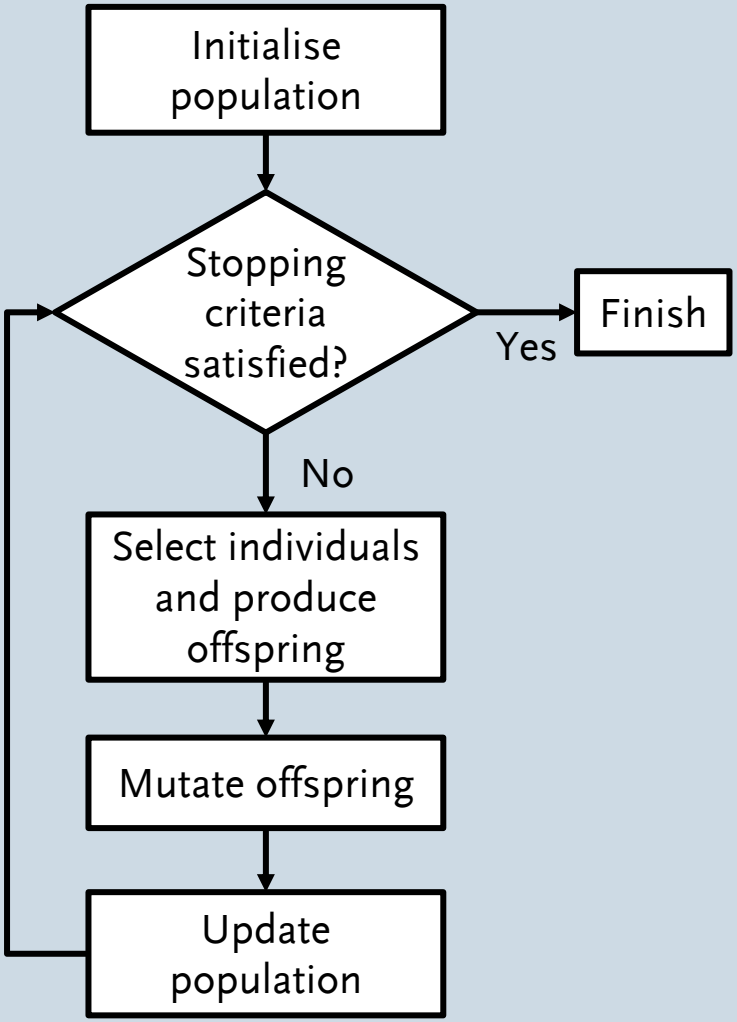
If $found.w < w_{new}$, then $found := \langle C, w_{new} \rangle$ end if

End for

return $found.C$

Description:
Deployment Level (Genetic Algorithm)

Genetic Algorithm (GA)



Stopping criteria

- Insignificant fitness improvement
- Maximum number of generations
- Elapsed time

Updating of population

- Keep best individuals and include offspring (i.e., elitism or steady state)

Components

- Genes, fitness function, initial population
- Selection, crossover, and mutation operators

Parameters

- Crossover and mutation probability
- Proportion of elite individuals, population size, and number of generations

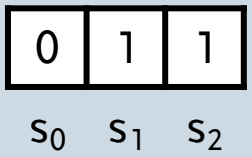
Contents

- Genes of an individual's chromosome
- Fitness function
- Initial population as solutions to a constraint program
- Genetic operators

Genes of an Individual's Chromosomes

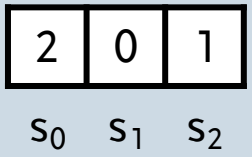
Each individual is a chromosome with four genes:

1. Selection of buffering protocol per signal



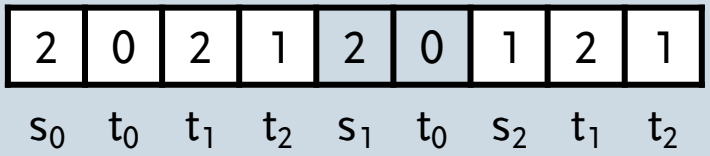
Selection for SBP is a binary value.
PTP = !SBP

2. SBP buffer-to-memory allocation



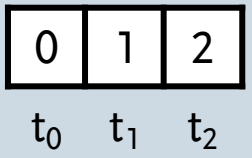
Memory module allocation is an integer value from 0 to |M|-1

3. PTP buffer-to-memory allocation



Memory module allocation is an integer value from 0 to |M|-1

4. Task-to-core allocation

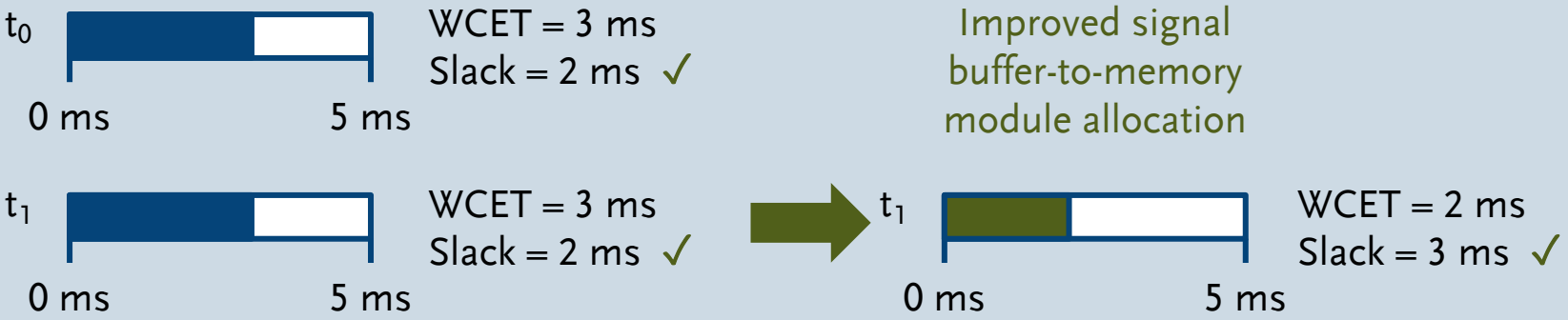


Core allocation is an integer value from 0 to |C|-1

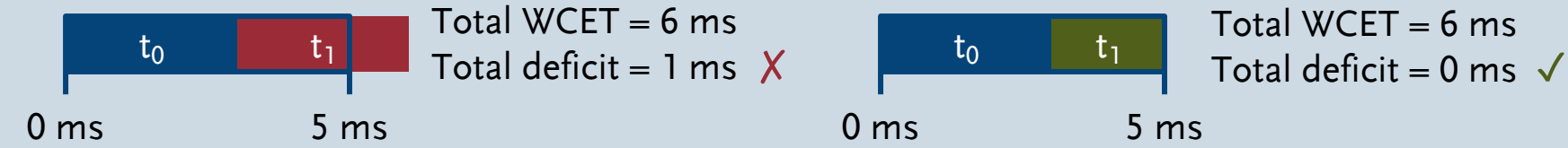
Fitness Function: Intuition

Deploying tasks t_0 and t_1 onto a single-core processor

Tasks:



Hyper period slot:



Fitness:

- Penalise total deficit (schedulability)
- Reward total slack (processor utilisation)

$$f(\psi) = \left(1 - \frac{1}{6}\right) * (2 + 2) = 3.\dot{3}$$

$$f(\psi) = (1 - 0) * (2 + 3) = 5$$

Fitness Function

For a system configuration ψ_i, the fitness function $f(\psi_i)$ calculates its schedulability and the slack of all tasks. A higher fitness value is better:

$$f(\psi_i) = Schedulaility(\psi_i) * \sum_{t \in T} \left(t.let * \frac{hp}{t.period} - \sum_{slot \in t_i.slots} (slot.t_i.alloc) \right)$$

where, $t.let = t.letEnd - t.letStart$, and $Schedulablility(\psi_i)$ reflects the degree of task schedulability, i.e., the proportion of task WCETs that can be scheduled.

Fitness Function

The $Schedulablility(\psi_i)$ function requires the calculation of the task execution times that exceed the duration of their allocated hyper period slots. The excess execution time is called a "deficit".

We find the minimum possible deficit with an MILP formulation. As before, task WCETs are allocated to the hyper period slots:

$$\forall t \in T, \forall t_i: \sum_{slot \in t_i.slots} (slot.t_i.alloc) \geq t.wcet_b + t.wcet_c$$

We use the real (positive) variable $deficit.c.slot$ to track the deficit of a hyper period $slot$ on core c:

$$deficit.c.slot \geq 0$$

$$\forall c \in C, \forall slot \in Slots: deficit.c.slot + slot.d - t_i.c.slot.alloc \geq 0$$

Fitness Function

The objective of the MILP is to minimise the total deficit:

$$Minimise\colon\ deficit.total = \sum_{c \in C} \sum_{slot \in Slots} deficit.c.slot$$

The total deficit is then used in the $Schedulablility(\psi_i)$ function:

$$Schedulablility(\psi_i) = 1 - \frac{deficit.total}{\sum_{t \in T} t.wcet}$$

Thus, we have the bound $0 \leq Schedulablility(\psi_i) \leq 1$, where 0 means that no tasks in ψ_i 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.

INITIAL POPULATION

- Simplification of the MILP formulation

Initial Population: Constraint Program (CP)

Solve a simplification of the MILP formulation

- Set of feasible solutions → Initial population for our GA
- MILP simplified into a constraint program (CP)
  - No objective function
  - Assume homogeneous cores with execution speed equal to the average of all the original cores: One WCET value per task, buffer management overhead, and context-switch, e.g.,

$$t.wcet_{instr} = \frac{\sum_{c \in C} t.instr.c.wcet}{|C|}$$

  - Assume uniform core-to-memory latencies: Shortest latency
  - Assume task utilisation is $t.wcet/t.period$: Optimistic approximation when $(t.letEnd - t.letStart) < t.period$
  - Assume maximum buffer management overhead for each signal, regardless of whether the signal uses SBP or PTP

Initial Population: Constraint Program (CP)

Worst-case buffer management time:

The worst-case time to manage task t's buffers is tracked by the (positive) real constant $t.wcet_b$. It is the sum of the maximum buffer management time due to SBP or PTP:

$$\forall t \in T\colon\ t.wcet_b = 2 * wcet_{cs} + \sum_{(s,n)\in t.acc} \left(\max\left(wcet_{sbp}, wcet_{ptp} + 2 * path * s.w\right)\right)$$

Worst-case computation time:

The worst-case time to execute task t's computations is tracked by the (positive) real constant $t.wcet_c$. It is the sum of the worst-case time to execute the instructions ($t.wcet_{instr}$), the context-switching time ($wcet_{cs}$), and the time to perform all buffered signal accesses:

$$\forall t \in T\colon\ t.wcet_c = t.wcet_{instr} + wcet_{cs} * t.cs + \sum_{(s,n)\in t.acc} (path * s.w * n)$$

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

$$\forall t \in T\colon t.wcet_b + t.wcet_c \leq t.letEnd - t.letStart$$

Initial Population: Constraint Program (CP)

Buffers

The Boolean variables $s.sbp$ and $s.ptp$ indicate whether SBP ($s.sbp = 1$) or PTP ($s.ptp = 1$), respectively, is used for signal s:

$$\forall s \in S\colon\ s.sbp + s.ptp = 1$$

SBP:

If SBP is used, the Boolean variable $sbp_s.m$ indicates whether the SBP buffer for signal s is allocated to memory module m ($sbp_s.m = 1$). The buffer can only be allocated to one memory module:

$$\forall s \in S\colon \sum_{m \in M} sbp_s.m = s.sbp$$

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

$$\forall m \in M\colon\ m.sbp = \sum_{s \in S} (sbp_s.m * s.size * s.n)$$

Initial Population: Constraint Program (CP)

PTP:

If PTP is used, the Boolean variable $ptp_s.t.m$ indicates whether task t's PTP buffer for signal s is allocated to memory module m ($ptp_s.t.m = 1$). Moreover, the Boolean variable $ptp_s.m$ indicates whether the signal s itself is allocated to memory module m ($ptp_s.m = 1$). Each buffer can only be allocated to one memory module:

$$\forall t \in T, \forall (s,n) \in t.acc: \sum_{m \in M} ptp_s.t.m = s.ptp$$

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

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

$$\forall m \in M: m.ptp = \sum_{t \in T} \sum_{(s,n) \in t.acc} (ptp_s.t.m * s.size) + \sum_{s \in S} (ptp_s.m * s.size)$$

Finally, the total utilisation of a memory module m due to SBP and PTP is limited by its size:

$$\forall m \in M: m.ptp + m.sbp \leq m.size$$

Initial Population: Constraint Program (CP)

Tasks

The Boolean variable $t.c$ indicates whether task t is allocated to core c ($t.c = 1$). A task can only be allocated to one core:

$$\forall t \in T: \sum_{c \in C} t.c = 1$$

Schedulability

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

$$\forall c \in C: \sum_{t \in T} \left(t.c * \frac{t.wcet_b + t.wcet_c}{t.period} \right) \leq 1$$

Initial Population: Constraint Program (CP)

Summary of notations used in the simplified constraint program

Variable Type	Notations
Boolean	$s.sbp, s.ptp, sbp_s.m, ptp_s.m, ptp_s.t.m, t.c$
Integer	$m.sbp, m.ptp$
Real	$t.wcet_b, t.wcet_c$

Constant Type	Notations
Integer	$s.size, s.n, s.w, t.acc.n, m.size, t.cs$
Real	$wcet_{cs}, wcet_{sbp}, wcet_{ptp}, path,$ $t.wcet_{instr}, t.letStart, t.letEnd$

GENETIC OPERATORS

- Selection operator
- Crossover operators
- Mutation operators

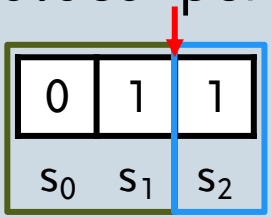
Selection Operator

Stochastic sampling to find pairs of parents

- Tournament:
  - Randomly select k-number of individuals, and then select the fittest individual
  - Increasing k increases the selection pressure
- Roulette/Fitness proportionate:
  - Chance of selecting an individual depends on its fitness relative to others

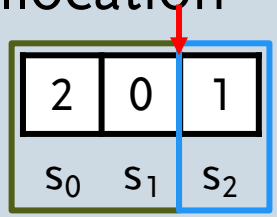
Crossover Operators

1. Selection of buffering protocol per signal



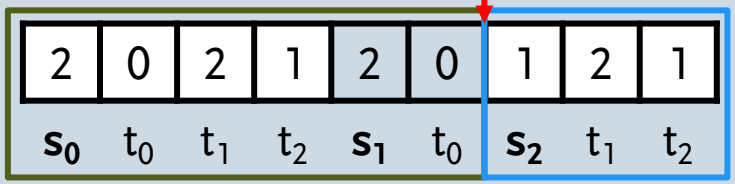
Any position within the gene

2. SBP buffer-to-memory allocation

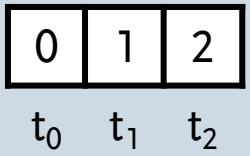


Crossover point for buffering protocol selection must also be used for the SBP and PTP genes

3. PTP buffer-to-memory allocation



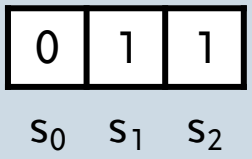
4. Task-to-core allocation



Any position within the gene

Mutation Operators

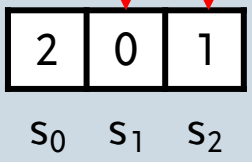
1. Selection of buffering protocol per signal



Selection for SBP is a binary value.
PTP = !SBP

Randomly negate the signal's selection

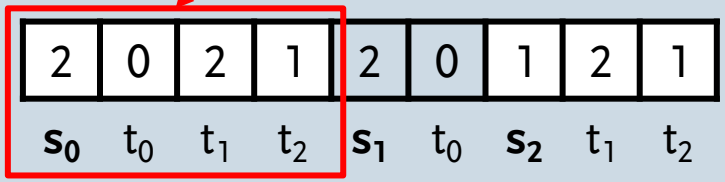
2. SBP buffer-to-memory allocation



Memory module allocation is an integer value from 0 to |M|-1

If a signal uses SBP, randomly set its value from 0 to |M|-1

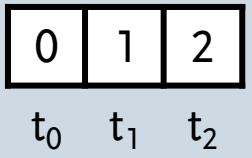
3. PTP buffer-to-memory allocation



Memory module allocation is an integer value from 0 to |M|-1

If a signal uses PTP, randomly set its value from 0 to |M|-1

4. Task-to-core allocation

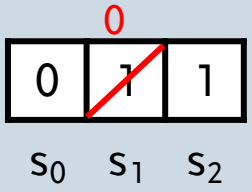


Core allocation is an integer value from 0 to |C|-1

Randomly set its value from 0 to |C|-1

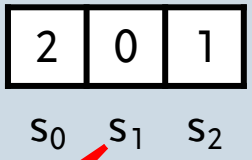
Mutation Operators

1. Selection of buffering protocol per signal

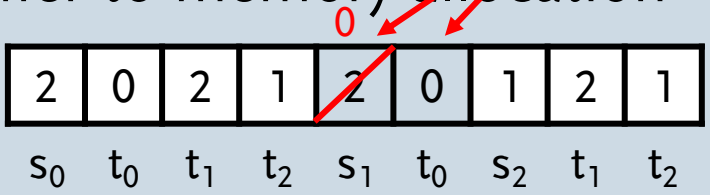


Selection for SBP is a binary value.

2. SBP buffer-to-memory allocation



3. PTP buffer-to-memory allocation



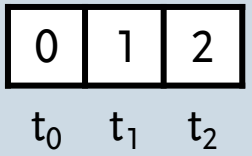
When changing a signal's buffering protocol, "repair" its buffer location:

- SBP to PTP: Set the PTP buffers to the same location as that of the SBP buffer
- PTP to SBP: Set the SBP buffer to same the location as that of the global signal's PTP buffer

an integer value from 0 to |M|-1

If a signal uses PTP, randomly set its value from 0 to |M|-1

4. Task-to-core allocation



Core allocation is an integer value from 0 to |C|-1

Randomly set its value from 0 to |C|-1