



# Segment Streaming for the Three-Phase Execution Model: Design and Implementation

Authors:

Muhammad R. Soliman  
Giovani Gracioli  
Rohan Tabish  
Rodolfo Pellizzoni  
Marco Caccamo

A decorative horizontal bar with a teal segment on the left and an orange segment on the right.

# Outline

- Introduction & Motivation
- Streaming Execution Model
- OS Programming Interface
- Task Set Segmentation
- Evaluation
- Conclusion & Future Work

A short horizontal bar with a teal segment on the left and an orange segment on the right.

# Introduction & Motivation

- Emerging Real-Time Applications
- Three-Phase Model
- Task Segmentation
- Example
- Previous Work Limitations
- Paper Contribution

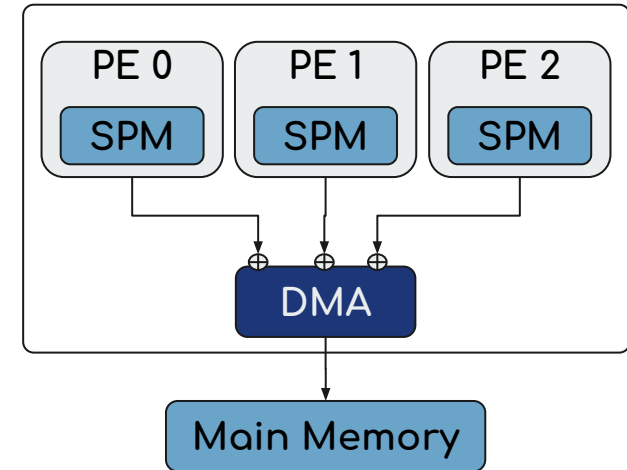


# Emerging Real-Time Applications

- **Requirements:** high performance, timing guarantees, and usually high usage of I/O devices (such as cameras and LIDAR sensors) and memory requirements.
- **Platforms:** Multiprocessor Systems-on-Chip (MPSoC) with a feature-rich environment, composed of multiple processing elements, high-bandwidth I/O devices and memories, and DMA engines.
- **Challenges:** shared resources that represent bottlenecks in terms of performance and predictability.

# Three-Phase Model

- Divides the task execution into 3 phases:
  - **LOAD:** Load the required code and data to SPM
  - **COMPUTE:** Execute the task directly from SPM
  - **UNLOAD:** Copy back the modified data
- Avoids contention at the main memory by sequentializing the load and unload phases of different processors.
- In a multiprocessor system, each processor is assumed to have a local memory (SPM/cache) and the code/data is move to/from the local SPM to the main memory during LOAD/UNLOAD phases. Arbitration techniques are used to share the memory bandwidth (e.g. TDMA).
- Double-buffering techniques allow overlapping the execution of one task while the DMA loads the memory of the next task by partitioning the local memory into two partitions.

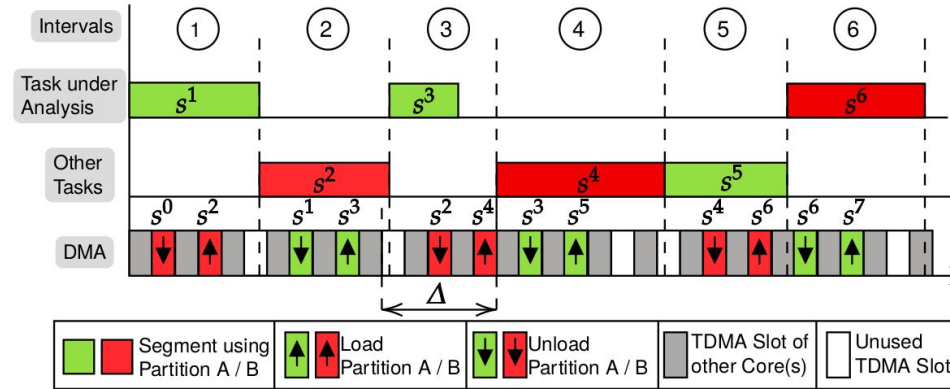




# Task Segmentation

- Practical implementation of the three-phase model requires program segmentation due to:
  - Size of local memory
  - Conditional program execution
  - Dynamic memory usage
  - Scheduling constraints
- A task is divided into multiple segments, each consisting of LOAD-COMPUTE-UNLOAD phases.
- We introduced an automated framework, based on the LLVM compiler, that segments a set of real-time tasks with the objective of improving the system's schedulability.
  - M. R. Soliman and R. Pellizzoni, "PREM-based Optimal Task Segmentation under Fixed Priority Scheduling," in 2019 31th Euromicro Conference on Real-Time Systems (ECRTS), July 2019

# Example



- Task under analysis has 3 segments ( $s_1, s_3, s_6$ ), segments ( $s_2, s_4, s_5$ ) belong to other tasks.
- During each interval, a segment of a task (ex:  $s_2$  in interval 2) executes using data and instructions in one partition. At the same time, the DMA unloads the content of the previous segment ( $s_1$ ) and loads the next segment ( $s_3$ ) in the other partition.



# Previous Work Limitations


- Previous work suffers from two main limitations:
  - Lack of the interface between the application and the OS to communicate the data to load/unload.
  - Memory latency can only be hidden by overlapping the memory and execution phases for segments of different tasks.
- These limitations can lead to schedulability degradation for systems that run few, or even a single task on each processor; which is common as the number of processors in an MPSoC increases.





# Paper Contributions

- Introduce the new streaming execution model to allow overlapping of memory and execution phases of segments of the same task.
- Extend the compiler framework to automatically segment streaming tasks.
- Propose an OS-independent application programming interface (API) to realize the scheduling and segmentation.
- Implement the API on a commercial RTOS and evaluate it using a latest-generation MPSoC.

A horizontal bar with a teal segment on the left and an orange segment on the right.

# Streaming Execution Model

- System Model
- Task Model
- Streaming Notation
- Execution Model
- Example



# System Model

- Sequential, sporadic three-phase tasks
- MPSoC platform comprising multiple processors.
- Tasks are partitioned to processors. Each processor has a dedicated SPM, which is used to execute its assigned tasks.
- A DMA engine is used to execute memory phases and shared among all processors.
- Fixed-time DMA model so that the time required to complete all memory phases in each scheduling interval is constant.
- SPM-based OS is used to control the schedule of task execution segments and memory phases.



# Task Model

- Task  $\tau_i$  Sporadic with period  $T_i$  and deadline  $D_i$  where  $D_i \leq T_i$
- Each job executes a sequence of segments that can vary during run-time.
- Program executed has a single entry and a single exit.
- Task is modeled as a Directed Acyclic Graph (DAG) of segments created by the compiler.
- Each DAG has a set of maximal paths, where a maximal path  $P$  is an ordered sequence of segments that starts from the entry segment and ends at the exit segment.

# Streaming Notation

- Each segment  $s$  is either a **terminal** or a **streaming** segment.
- A segment  $s_j$  is a **streaming** segment if  $s_j$  has a unique immediate successor segment  $s_k$ , and  $s_k$  can be executed in the scheduling interval after the one where  $s_j$  executes; otherwise,  $s_j$  is **terminal**.

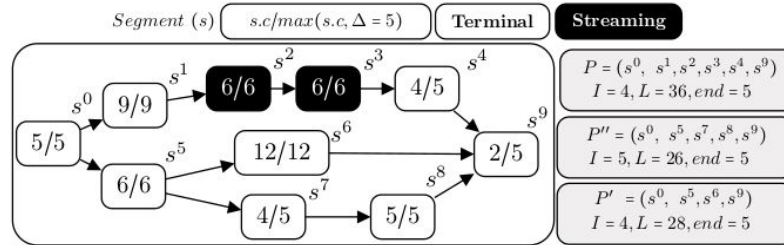


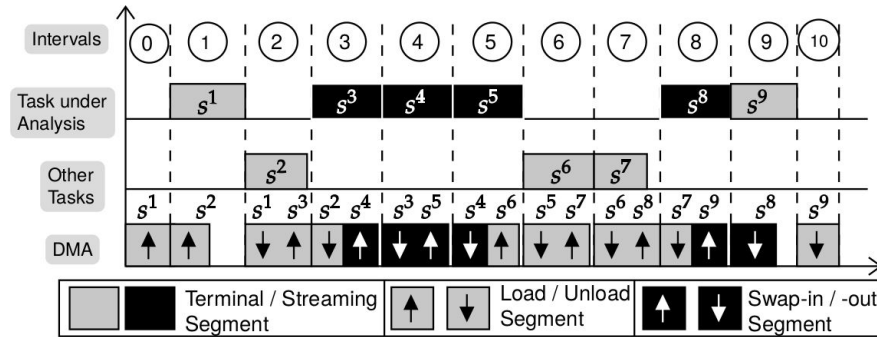
Fig. 2: Example segment DAG ( $s^0$  is  $s^{begin}$  and  $s^9$  is  $s^{end}$ ).




# Execution Model

- The SPM is not necessarily divided into two partitions, but footprint of each segment (number of virtual memory pages) is no larger than half the SPM size.
- DMA operations depends on the type of segments :
  - **Terminal**: same as previous work, LOAD/UNLOAD operations.
  - **Streaming**: SWAP-IN/SWAP-OUT operations, where only the pages that changes between segments are unloaded/loaded.

# Example



- Intervals 0, 1, 2 have **terminal** segments and follows the three-phase model.
- Intervals 3, 4, 5 show the case of multiple **streaming** segments.
- Interval 6 shows the case of preempting the **streaming** segments from higher priority tasks.
- Interval 8 shows the case of resuming segment **streaming** after preemption.
- Interval 9 switches from **streaming** to **terminal** segments.

A short horizontal bar with a teal segment on the left and an orange segment on the right.

# OS Programming Interface

- Goal
- API Description
- Example





# Goal

- The goal is to have a general OS API to be inserted in the code of a task in order to:
  - Partition the task into segments
  - Manage the SPM content.
- The API is generic for the three phase model and can be used manually or automatically during compilation.

# API Description

- The SPM allocated to a task is partitioned into a set of buffers and objects that are allocated/deallocated using API calls:
  - A buffer is used for segment streaming such that the content of the buffer can be swapped during execution.
  - Other SPM allocations that are not buffers are called objects.

<code>int allocate(uint64_t *src, uint64_t *dst, int size, int attr) /</code>
<code>int allocate2d(uint64_t* src, uint64_t* dst, int width, int height, int spitch, int dpitch, int attr) :</code>
Allocate an object at <code>dst</code> and copy 1D/2D array from <code>src</code> if <code>attr</code> is RO/RW → return the ID assigned to the object.
<code>void deallocate(int id) :</code> Release the object with ID <code>id</code> and write-back the data if the object is WO/RW.
<code>int allocate_buffer(uint64_t *dst, int attr) :</code> Allocate a buffer at <code>dst</code> for data streaming → return the buffer ID.
<code>void swap_buffer(int id, uint64_t *src, int size) /</code>
<code>void swap2d_buffer(int id, uint64_t *src, int width, int height, int spitch, int dpitch) :</code>
Swap the 1D/2D data in the buffer with ID <code>id</code> by writing-back the current data for WO/RW buffer and copying data from <code>src</code> for RO/RW buffer.
<code>void deallocate_buffer(int id) :</code> Release the buffer with ID <code>id</code> and write-back the data if the buffer is WO/RW.
<code>void dispatch() :</code> Force all buffer DMA requests to move from waiting queue to dispatch queue.
<code>void end_segment() :</code> Ends segment execution.

# Example

```
int h[128];
char a[500];

void init(int* x, int n) {
    for(int i = 0; i < n; i++)
        x[i] = 0;
}

void hist(int *x, int *y, int n){
    for(int i = 0; i < n; i++) {
        y[i] = x[i] & 127;
        x[y[i]] += 1;
    }
}

void histogram() {
    init(h, 128);
    hist(h, a, 500);
}
```

(a) Original code.

```
void histogram() {
    // 3PT is initialized to allocate
    // h to h_obj with ID (01)

    init(h_obj, 128)
    B1 = allocate_buffer(a_buf1, RW);
    B2 = allocate_buffer(a_buf2, RW);
    swap_buffer(B1, a, 100)
    dispatch();
    swap_buffer(B2, a+100, 100)
    end_segment();

    hist(h_obj, a_buf1, 100)
    swap_buffer(B1, a+200, 100);
    end_segment();

    hist(h_obj, a_buf2, 100)
    swap_buffer(B2, a+300, 100);
    end_segment();

    hist(h_obj, a_buf1, 100)
    swap_buffer(B1, a+400, 100);
    end_segment();

    hist(h_obj, a_buf2, 100)
    deallocate_buffer(B2);
    end_segment();

    hist(h_obj, a_buf1, 100)
    deallocate(01);
    deallocate_buffer(B1);
    end_segment();
}
```

(b) Segmented code.

A short horizontal bar with a teal segment on the left and an orange segment on the right.

# Task Set Segmentation

- Task Set Segmentation Algorithm
- Program Segmentation Algorithm

# Task Set Segmentation Algorithm

- The algorithm recursively explores the task set from the highest to the lowest priority task. At each step  $i$ , the value of  $l_{max}$  represents the maximum length of segments of tasks  $(\tau_i, \dots, \tau_n)$  under which the previously explored higher priority tasks  $(\tau_1, \dots, \tau_{i-1})$  are schedulable.
- The algorithm ends if all tasks are segmented or exits if any task is not schedulable.
- Schedulability analysis provided in the paper proves the correctness of the algorithm.

---

## Algorithm 2 Task Set Segmentation

---

**Require:** Task set  $\Gamma$ , source code for each task in  $\Gamma$

```

1: SEGMENTTASKSET( $\Gamma, i, +\infty, \emptyset$ )
2: Terminate with FAILURE
3: function SEGMENTTASKSET( $\Gamma, i, l^{\max}, \{G_1, \dots, G_{i-1}\}$ )
4:   Generate  $\mathcal{G}_i = \text{SEGMENTTASK}(\tau_i, l^{\max})$ 
5:   if  $i < N$  then
6:     for all  $G_i \in \mathcal{G}_i$  do
7:       Compute the maximum value  $\overline{l_i^{l_{\max}}}$  of  $l_i^{\max}$ 
         based on analysis
8:       SEGMENTTASKSET( $\Gamma, i + 1,$ 
          $\min(l^{\max}, \overline{l_i^{l_{\max}}}), \{G_1, \dots, G_i\}$ )
9:   else
10:    for all  $G_N \in \mathcal{G}_i$  do
11:      If analysis returns schedulable on  $\{G_1, \dots, G_N\}$ , terminate
         with SUCCESS

```

---

# Program Segmentation Algorithm

- We adopt the segmentation algorithm from our previous work:
  - The compiler creates a tree-based representation of the program based on its CFG. Nodes in the tree, called program regions, represent either basic blocks, conditionals, loops, or function calls.
  - Program segmentation is equivalent to assigning regions to segments while maintaining the structure of the program, and respecting constraints on the segment length  $l_{max}$  and its maximum footprint.
  - Loop splitting and tiling are used to fit large data structures in the SPM, and break loops into shorter segment lengths.
- Two modifications on the algorithm based on the new model:
  - The tiling algorithm now returns a sequence of streaming segments.
  - The time of all API calls are accounted for in the algorithm.



# Evaluation

- API Implementation
- API Overhead
- Schedulability Evaluation
- Evaluation Cases
- Case 1 Results
- Case 2 Results
- Results Summary



# API Implementation

- The API and OS-related techniques were implemented in Erika Enterprise RTOS.
- The execution times and memory footprint of the API functions are compiled and measured using the Xilinx UltraScale+ ZCU102 MPSoC platform:
  - Four-core 1.2Ghz ARM Cortex-A53 processor.
  - 32KB local instruction and data caches / core.
  - Last-Level Cache (LLC) of 1MB shared by all the A53 cores.
  - DDR4-2666 main memory controller of 64-bit data width connected to a 4GB DDR4 memory module and a programmable logic.
- The segmented code of the tasks using LLVM after adding the API calls, then linked together with Erika using the gcc compiler version 7.2.1.



# API Overhead

	Code (bytes)	AVG	STD	WCET	BCET	Var. Window
<b>allocate_buffer</b>	76	46.24	29.02	949	40	0.96
<b>dispatch</b>	112	197.32	18.14	717	191	0.73
<b>DMA_int_handler</b>	136	82.81	29.00	989	80	0.92
<b>allocate</b>	268	255.67	31.85	1252	252	0.80
<b>end_segment</b>	284	79.11	47.24	1565	70	0.96
<b>deallocate</b>	300	182.32	93.07	717	50	0.93
<b>allocate2d</b>	308	262.83	20.84	919	262	0.71
<b>deallocate_buf</b>	320	256.18	140.52	646	60	0.91
<b>swap_buffer</b>	400	357.17	121.50	1595	262	0.84
<b>swap2d_buffer</b>	444	397.62	131.36	1040	282	0.73

- Code size (in bytes) and average (with standard deviation), worst-case, and best-case execution time (in ns) of the API implementation.

# Schedulability

- Proposed streaming execution model VS three-phase model without streaming.
- Ideal program segmentation is used as a reference for the best possible performance:
  - No restrictions on the SPM size.
  - The program code can be segmented at any arbitrary point without any overhead.
- API impact is assessed by measuring the schedulability with and without API Overhead.
- Benchmarks: (UTDSP, TACLeBench, and CortexSuite)

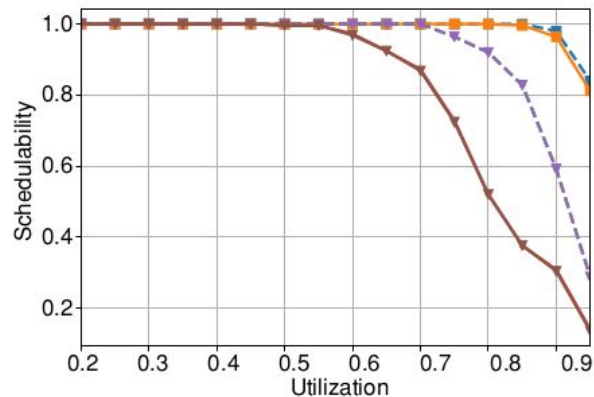
Benchmark	Suite	LOC	Data (bytes)	WCET (ns)
adpcm_dec	TACLeBench	476	404	176947
cjpeg_transupp	TACLeBench	474	3459	12083696011
fft	TACLeBench	173	24572	89540809
compress	UTDSP	131	136448	168984645
lpc	UTDSP	249	8744	233390
spectral	UTDSP	340	4584	109074793
disparity	CortexSuite	87	2704641	339361377

# Evaluation Cases

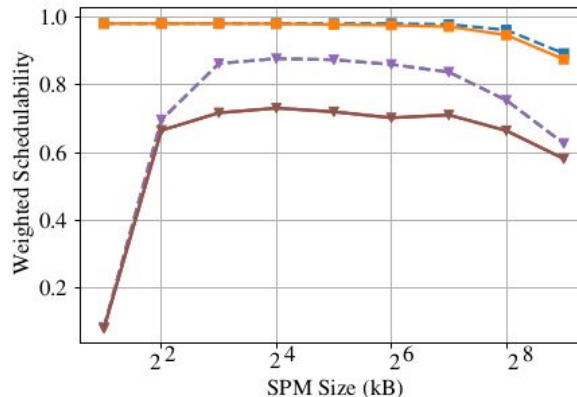
- We use the benchmarks to generate synthetic task sets for two cases:
  - Case 1: Random number of tasks between 5 and 15. The utilization of each task is uniformly generated given a system utilization and the number of tasks.
  - Case 2 (An example of a system that can benefit the most from the streaming model): Force the benchmark 'disparity' to be a heavy load task with 50% of the system utilization, while generating 3 to 5 tasks with the other 50% of the system utilization.
- System utilization varies between 0.2 and 0.95 using 250 task sets per utilization level.
- The SPM size varies between 2 kB and 512 kB.
- The DMA throughput varies between 0.1 GB/s to 4.0 GB/s
- Results are presented in terms:
  - System schedulability: the proportion of the schedulable task sets out of the total tested task sets.
  - Weighted schedulability metric:

$$\mu = \frac{\sum_u sched(u) \cdot u}{\sum_u u}$$

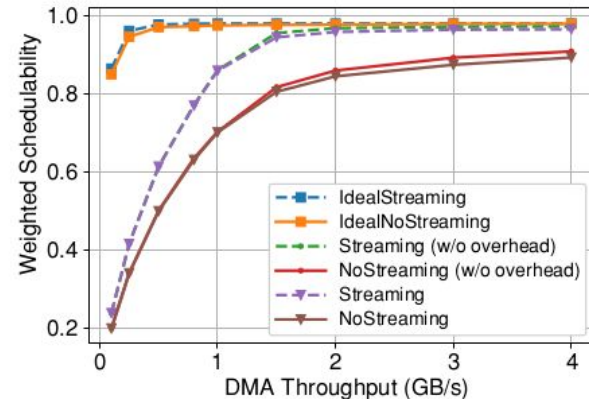
# Case 1 Results



(a) Schedulability vs Utilization  
(SPM size = 64 kB / DMA throughput = 1.0 GB/s)



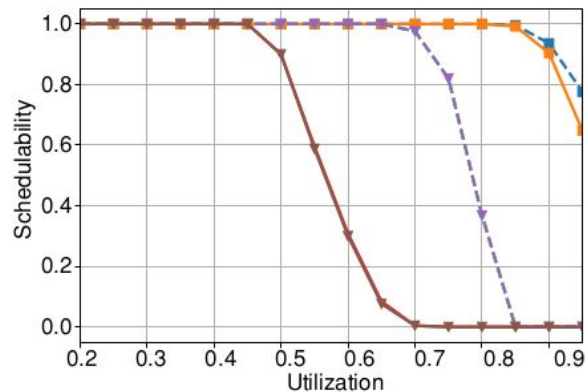
(b) Weighted Schedulability vs SPM size  
(DMA throughput = 1.0 GB/s)



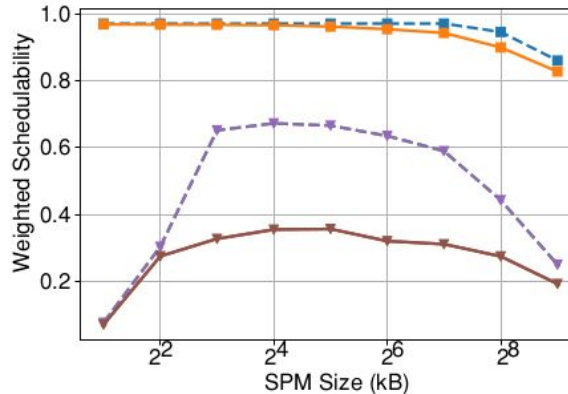
(c) Weighted Schedulability vs DMA throughput (SPM size = 64 kB)

Fig. 7: Case 1: uniformly generated random task sets.

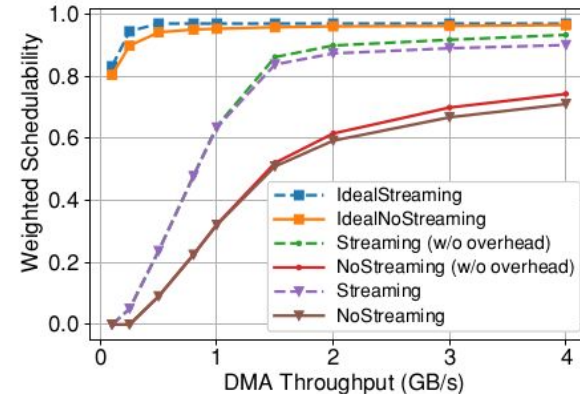
## Case 2 Results



(a) Schedulability vs Utilization  
(SPM size = 64 kB / DMA throughput = 1.0 GB/s)



(b) Weighted Schedulability vs SPM size  
(DMA throughput = 1.0 GB/s)



(c) Weighted Schedulability vs DMA throughput  
(SPM size = 64 kB)

Fig. 8: Case 2: heavy load task sets with ‘disparity’ using 50% of the system utilization.



# Results Summary

- Case 1: The streaming model results in an improvement up to 16% for the weighted schedulability compared the non streaming mode.
- Case 2: The streaming model results in an improvement up to 33% for the weighted schedulability compared the non streaming mode.
- The schedulability improves at the beginning as the SPM size increases since larger SPM sizes cause less segmentation and hence less overhead and less blocking from lower priority tasks. Then, the performance peaks and starts to decline as large SPM sizes imply large DMA times due to the fixed-time assumption used in the analysis; this causes under-utilization of the segments with smaller execution time.
- Increasing the DMA throughput improves the schedulability as the impact of the DMA time on the response time of the tasks is reduced.
- The effect of the overhead on the schedulability is minimal and becomes only relevant at high DMA throughput when segment length is shorter.



# Conclusion & Future Work



# Conclusion

- The three-phase execution model has proven effective in increasing predictability of memory accesses while hiding access latency through the use of a DMA engine.
- We have proposed a new streaming model for three-phase tasks that allows consecutive segments of the same task to execute sequentially, improving schedulability.
- We have described an automatic compilation framework that adds all required OS calls to the programs, and evaluated the approach on a commercial RTOS and realistic benchmarks.





# Future Work

- Make the model more applicable by relaxing program constraints and provide automatic program transformation algorithms.
- Improve the schedulability by making more complex assumptions on DMA time (variable-time model).
- Extend the API to handle different I/O device types (synchronous and asynchronous) and hardware accelerators in a way compatible with the three-phase model