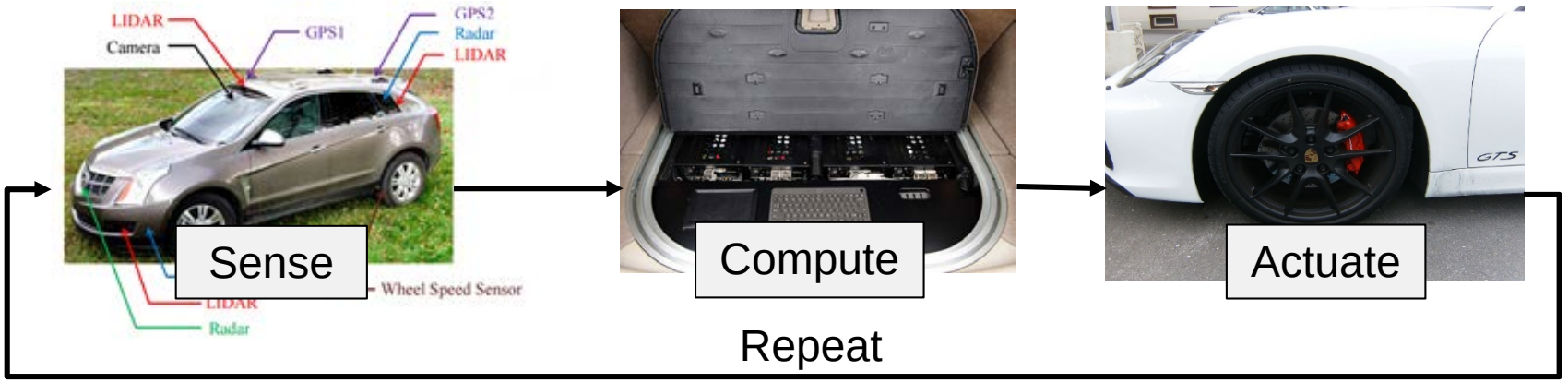
Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions

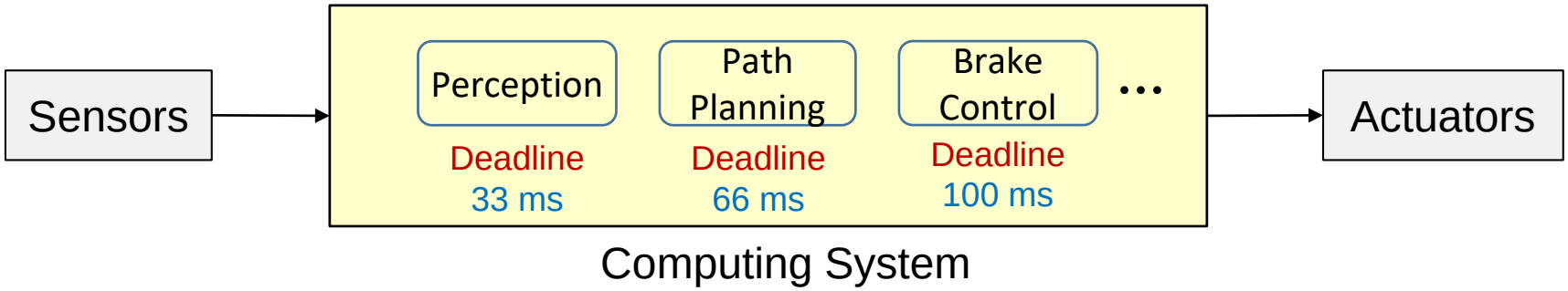
Pratyush Patel, Iljoo Baek, Hyoseung Kim*, Raj Rajkumar

Carnegie Mellon

* UC Riverside University of California

High Computational Demand of Safety-Critical Systems





- ✓ Long execution time
- ✓ Difficult to meet **deadlines**

} Need Computational Accelerators*†

* GP-GPUs (General-Purpose GPUs)
† Digital Signal Processor (DSP)

Problems with Hardware Accelerators

- They do not support preemption
  - Due to high context switching overhead*†

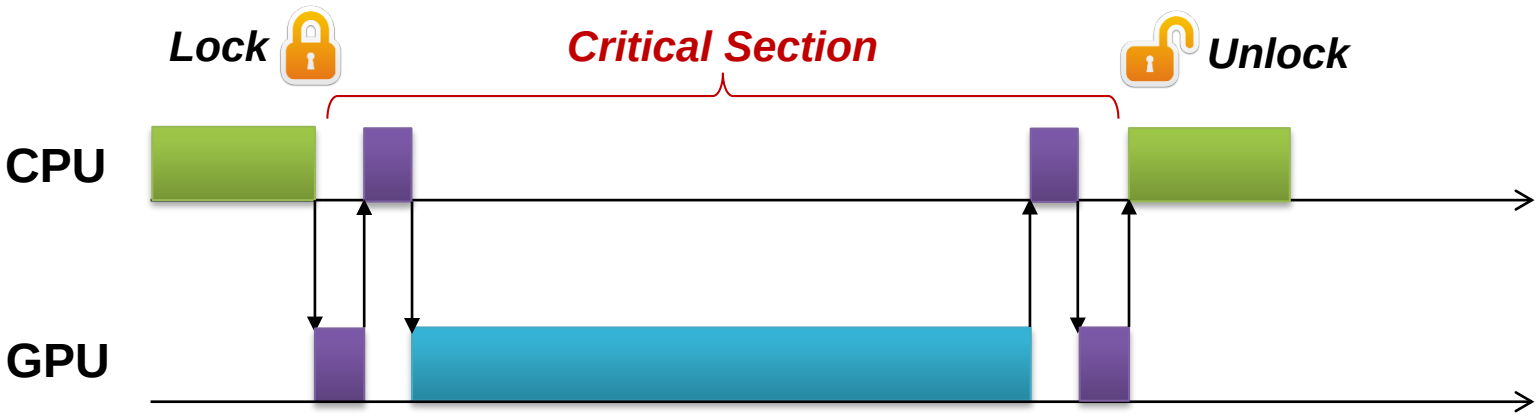
- They handle multiple resource requests in any order
  - Concurrent execution on GPU may result in unpredictable delays

3 identical CUDA kernels on NVIDIA GTX 1070
- ✓ 97% slowdown on two kernels
- ✓ Unpredictable which kernel gets delayed

- They do not respect task priorities or scheduling policies
  - May result in unbounded priority inversion

* I. Tanasicet al. Enabling preemptive multiprogramming on GPUs. In *International Symposium on Computer Architecture (ISCA)*, 2014.
† Some recent GPU architectures support preemption - NVIDIA Volta Architecture https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/

Existing Solution: Synchronization-Based Approaches*†‡



Benefits of synchronization-based approaches

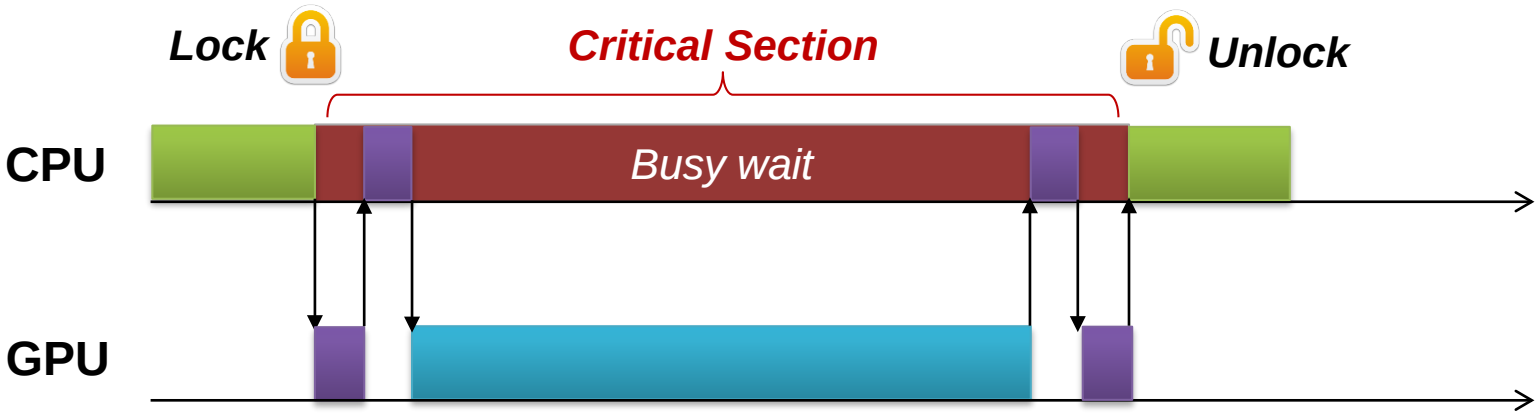
- ✓ Do not require any change in accelerator device drivers
- ✓ Existing schedulability analyses can be directly re-used

* G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Syst.*, 48(1):34–74, 2012.
† G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Syst.*, 49(2):140–170, 2013.
‡ G. Elliott et al. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

Limitations

- Busy waiting
  - Critical sections are executed entirely on the CPU

Common assumption of most RT synch. protocols, e.g., MPCP, FMLP†, OMLP‡*



- Analytical pessimism
  - Traditional recursion-based analysis#
  - Can lead to expensive over-provisioning

* R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1988.
† A. Block et al. A flexible real-time locking protocol for multiprocessors. In *IEEE Embedded and Real-Time Comp. Systems and Apps., (RTCSA)*, 2007.
‡ B. Brandenburg and J. Anderson. The OMLP family of optimal multiprocessor real-time locking protocols. *Design Automation for Embedded Systems*, 2013
# K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In IEEE Real-Time Systems Symposium (RTSS), 2009.

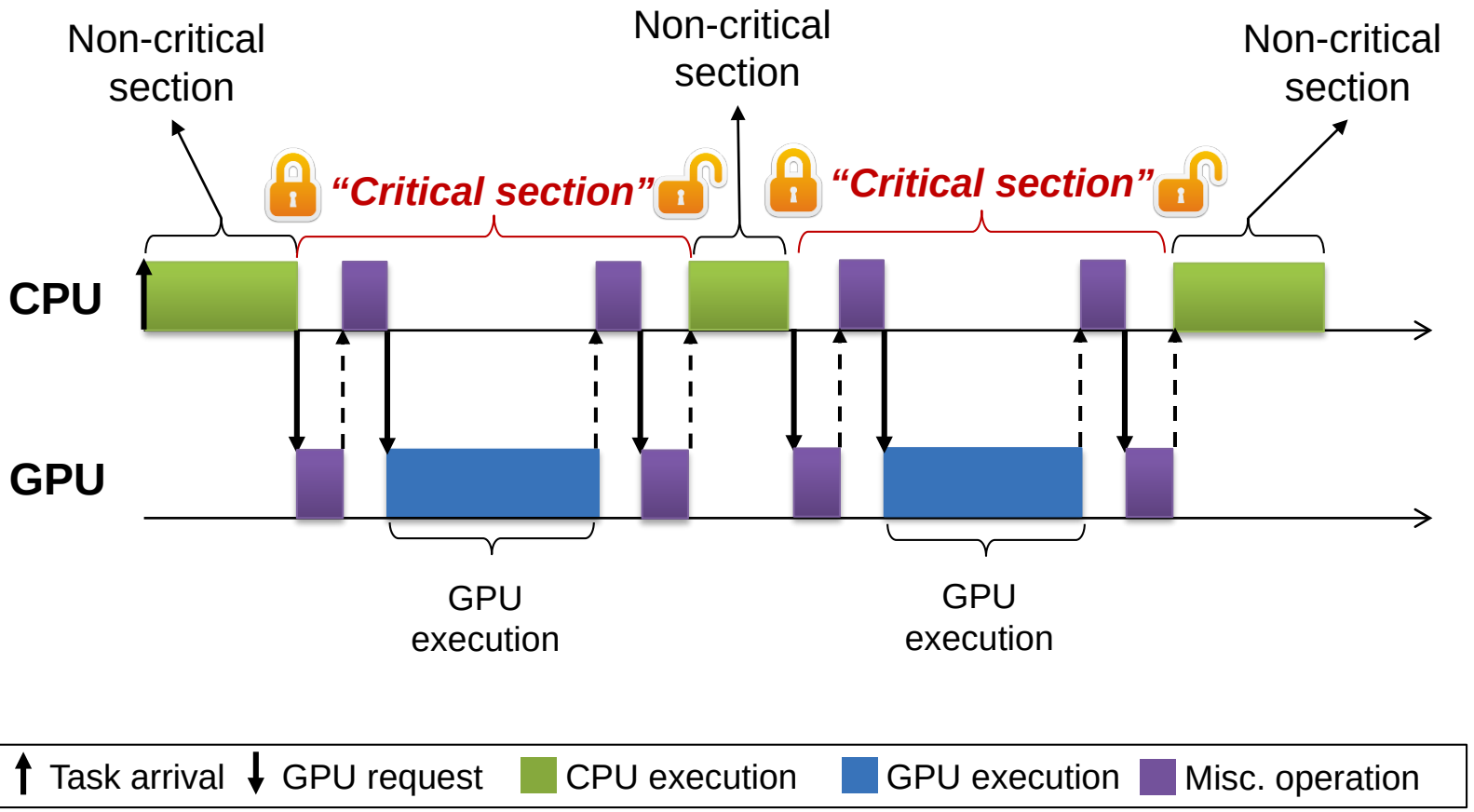
Our Contributions

- Analytical enhancements for the **Multiprocessor Priority Ceiling Protocol (MPCP)**
  - ✓ Tighter bounds for task response times
  - ✓ Allow suspensions when executing critical sections
- Extensive schedulability experiments for a variety of task set parameters
- Prototype implementation and evaluation on Nvidia TX2 running Linux
- Extensions can be used with multiple types of computational accelerators, such as a digital signal processor (DSP) and General-Purpose GPU (GP-GPU)

Outline

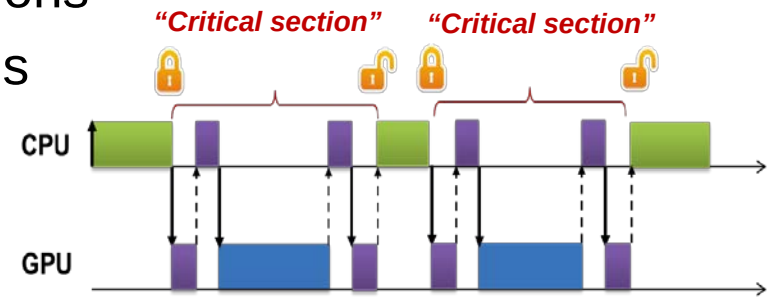
- Motivation & Introduction
- **Suspension-based MPCP**
  - System model
  - Comparison with busy-waiting approach
  - Task response time analysis
- Evaluation
- Conclusions

Example of GPU Execution



System Model

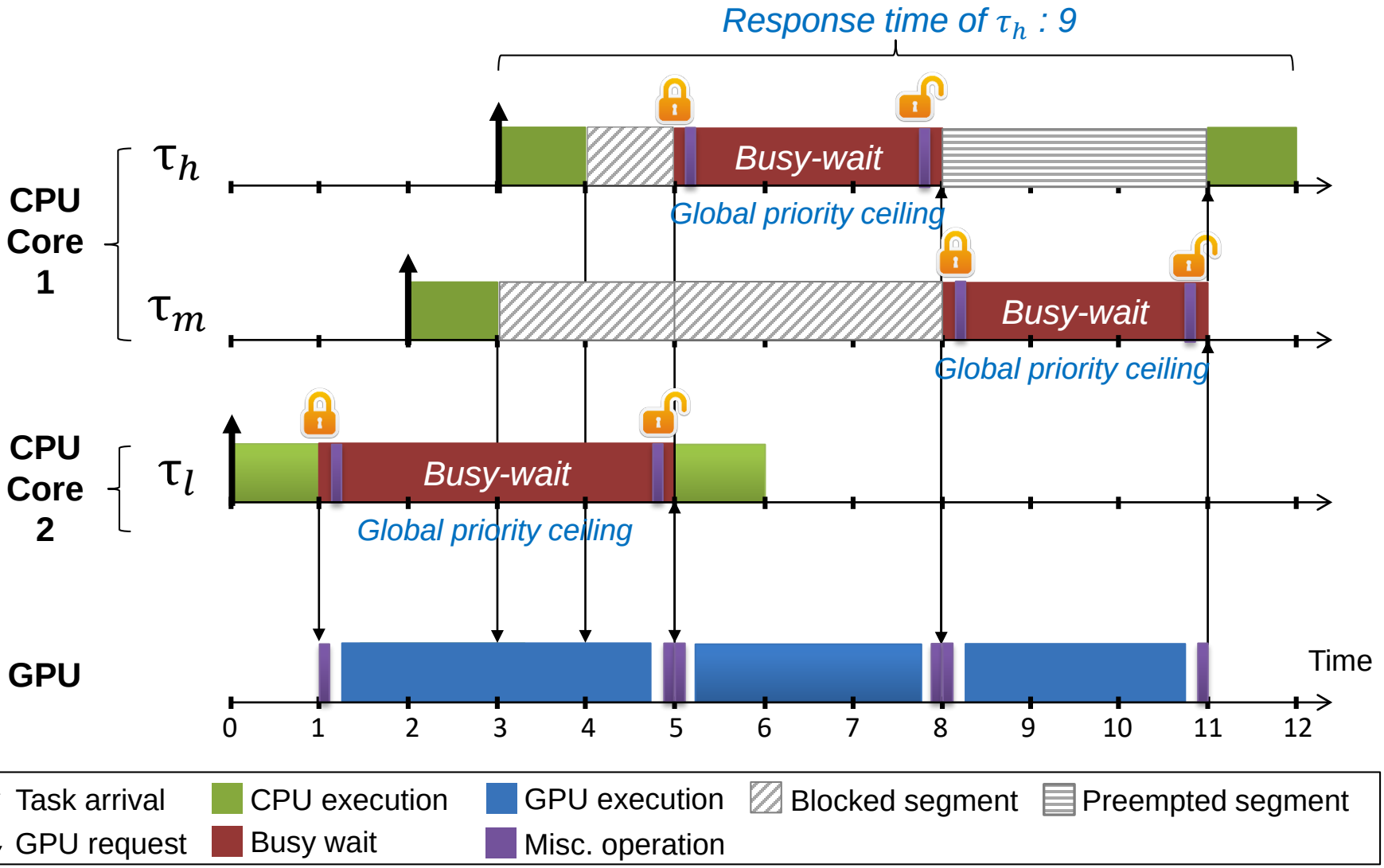
- Sporadic tasks with constrained deadlines
  - Task $\tau_i := (C_i, G_i, T_i, \eta_i)$
    - C_i : Sum of the WCET* of all non-critical sections
    - G_i : Sum of the WCET* of all critical sections
    - T_i : Period (Deadline = Period)
    - η_i : Maximum number of critical sections
    - $\zeta_{i,j}$: Maximum number of suspensions in the j[th] critical section
  - Critical segment $G_{i,j} := (G_{i,j}^e, G_{i,j}^m)$



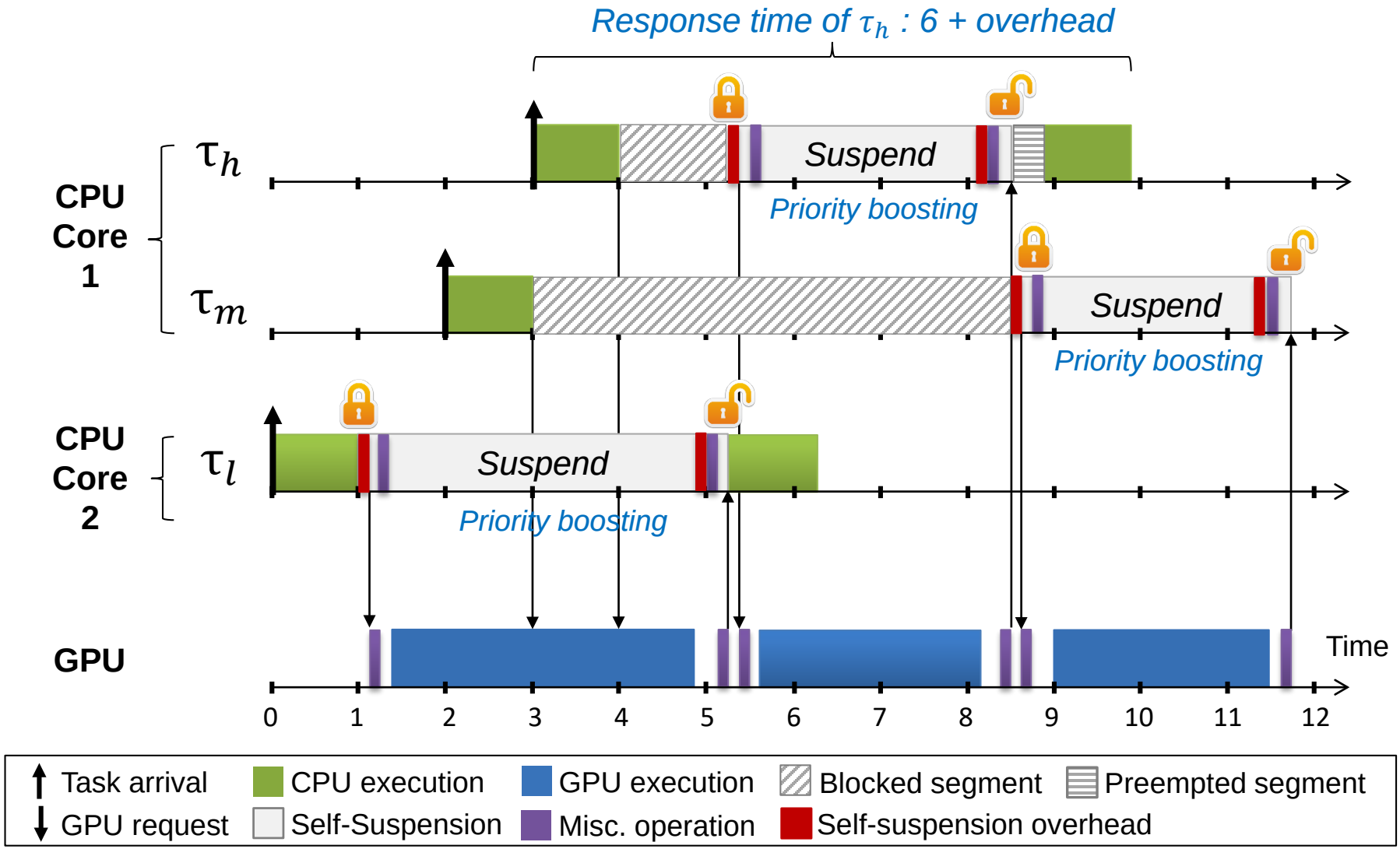
- Each hardware accelerator is modeled as a distinct shared resource
- Use partitioned fixed-priority preemptive scheduling

* Worst-Case Execution Time (WCET)

Example under *Busy-Waiting* MPCP



Example under Suspension-based MPCP



Task Response Time Analysis

- Worst-Case Response Time (W_i)* i = task number

Self-suspension by higher-priority tasks†

$$W_i^{n+1} = C_i + G_i + B_i + \sum_{\tau_h \in hpp(\tau_i)} \left\lceil \frac{W_i^n + W_h - C_h}{T_h} \right\rceil \cdot C_h$$

Blocking delay

Our contribution

Preemption delay by Higher-priority tasks

* N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal, 8(5):284–292, 1993.

† J.-J. Chen et al. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. Technical Report 854, Department of Computer Science, TU Dortmund, 2016.

Total Blocking Time Analysis

For each analyzed task,

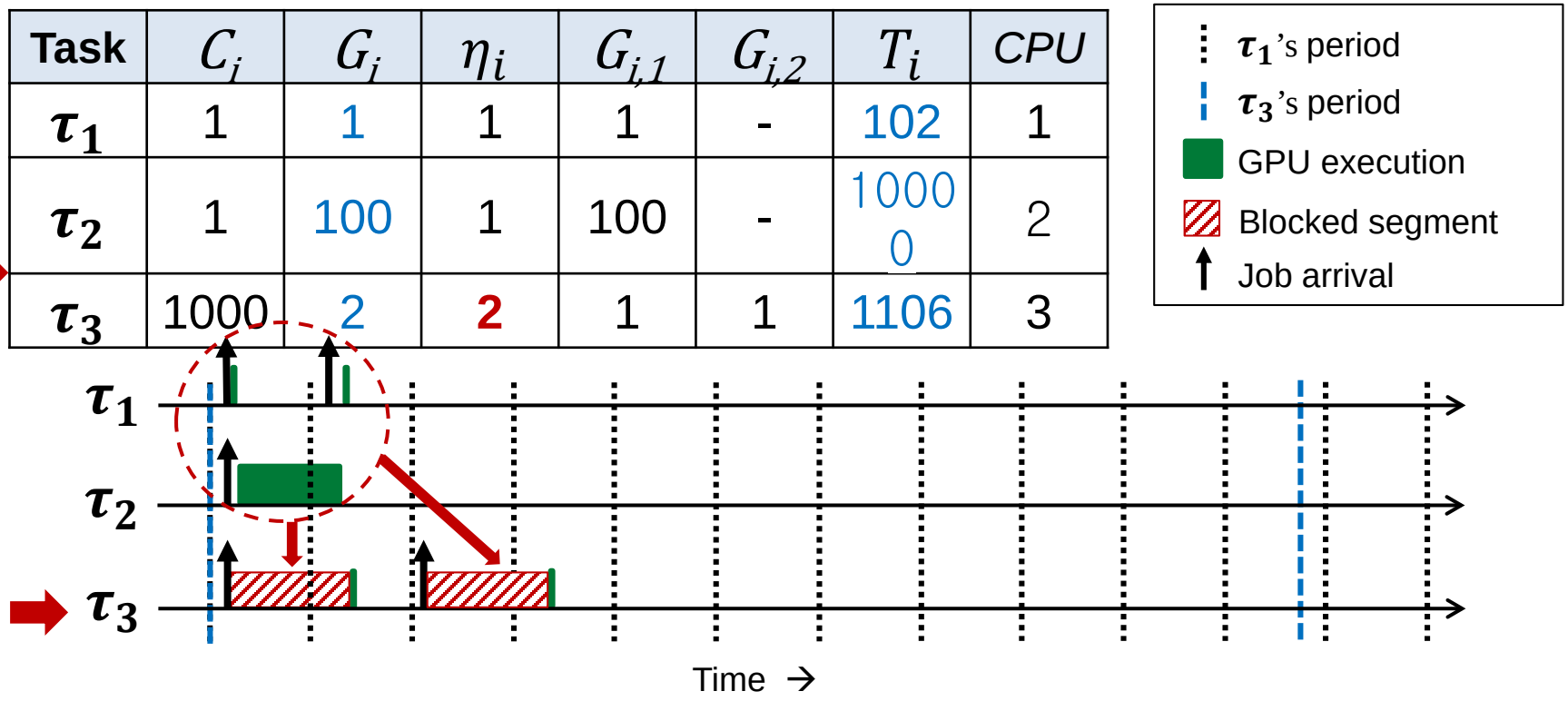
- **Request-driven (RD) Approach***
  - Consider the sum of the worst-case blocking times *for* *each lock-acquisition request issued by* ***the analyzed task***
- **Job-driven (JD) Approach**
  - Consider the maximum number of lock-acquisition requests issued by **other tasks** during the execution of the analyzed task
- **Hybrid Approach**
  - Upper-bound the maximum lock-acquisition requests possible in **RD** analysis by using **JD** analysis – obtain the best of both approaches
    - Different from RTAS'14† which simply takes the minimum of RD and JD for the blocking delay

* K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In IEEE Real-Time Systems Symposium (RTSS), 2009.

† H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

Request-driven*
Blocking Time Analysis

Task	C_i	G_i	η_i	$G_{i,1}$	$G_{i,2}$	T_i	CPU
τ_1	1	1	1	1	-	102	1
τ_2	1	100	1	100	-	10000	2
τ_3	1000	2	**2**	1	1	1106	3



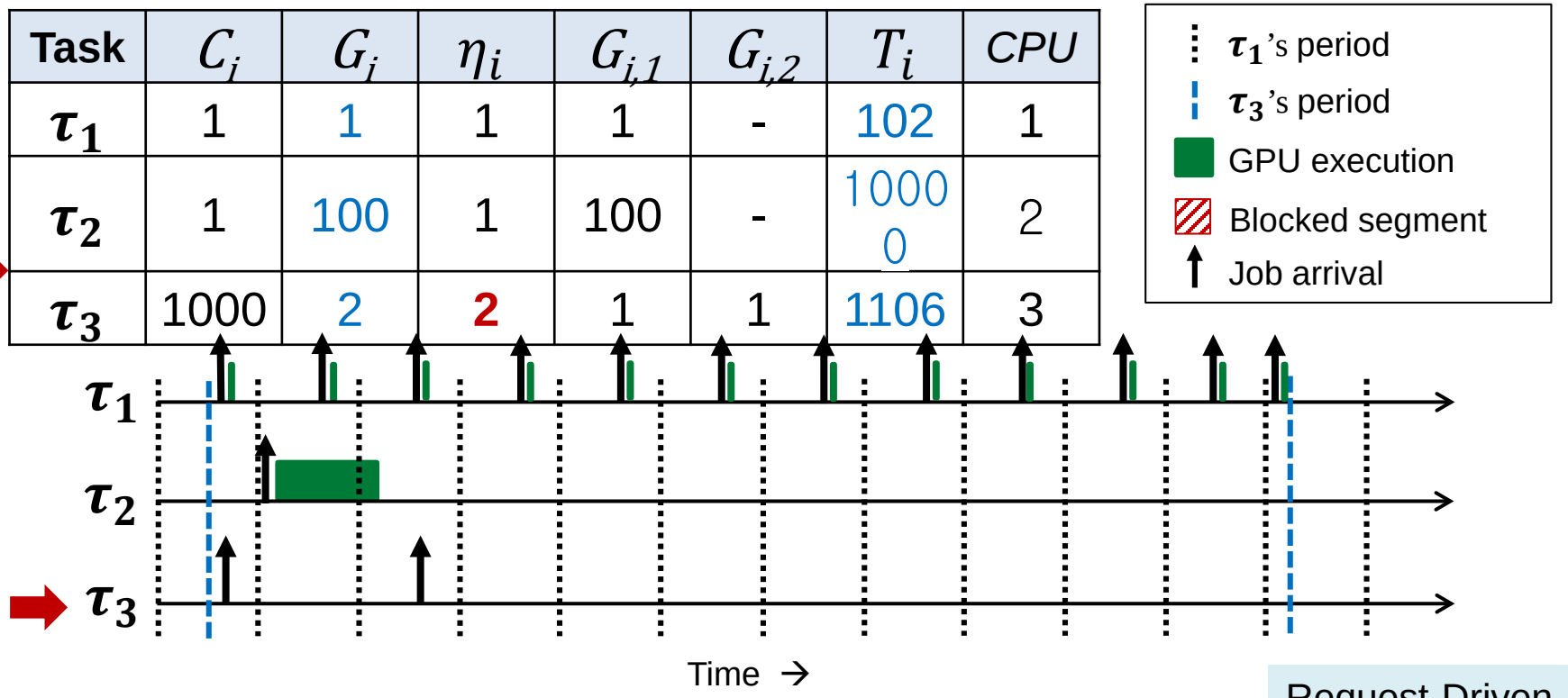
For each request made by τ_3, **its blocking time is given by**

$$B_3 = B_{3,1} + B_{3,2} \quad = 102 + 102 = \mathbf{204}$$

* K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In IEEE Real-Time Systems Symposium (RTSS), 2009.

Job-driven
Blocking Time Analysis

Task	C_i	G_i	η_i	$G_{i,1}$	$G_{i,2}$	T_i	CPU
τ_1	1	1	1	1	-	102	1
τ_2	1	100	1	100	-	10000	2
τ_3	1000	2	**2**	1	1	1106	3



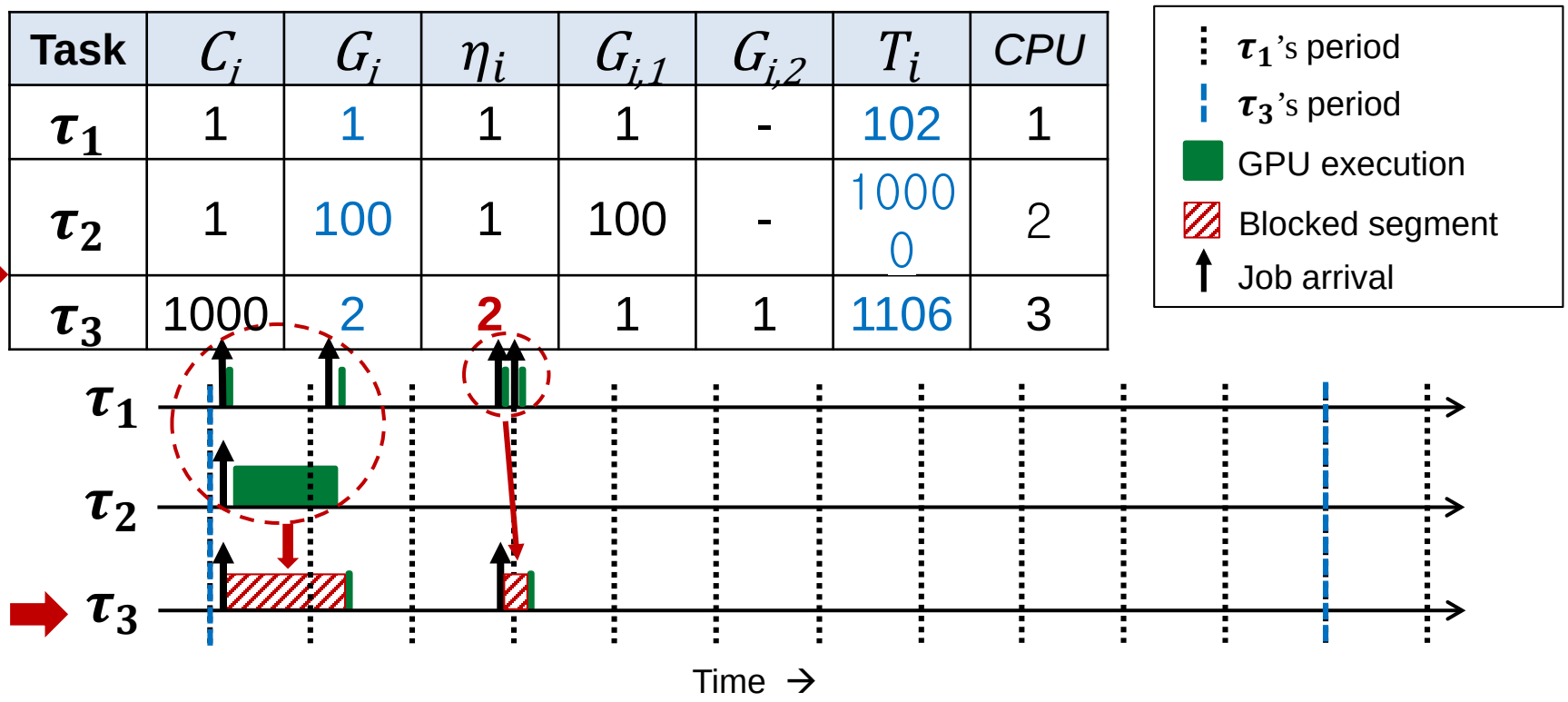
Request-Driven
B_3 = **204**

The **max requests (= jobs)** made by **other tasks**

# of req. by τ_1= 12 **# of req. by τ_2= 1** B_3 = 12x1 + 1x100 = **112**

Hybrid
Blocking Time Analysis

Task	C_i	G_i	η_i	$G_{i,1}$	$G_{i,2}$	T_i	CPU
τ_1	1	1	1	1	-	102	1
τ_2	1	100	1	100	-	10000	2
τ_3	1000	2	2	1	1	1106	3



Both **each req. made by τ_3** and **max req. made by other tasks**

$B_3 = B_{3,1} + B_{3,2}$ = (4x1) + (1x100) = **104**

Job-Driven
B_3 = **112**

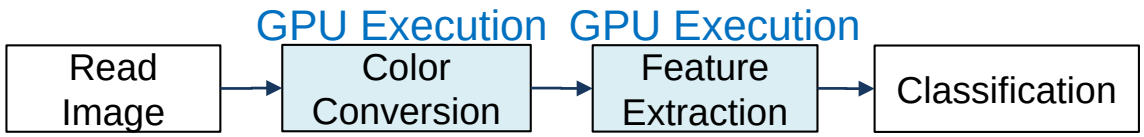
Request-Driven
B_3 = **204**

Outline

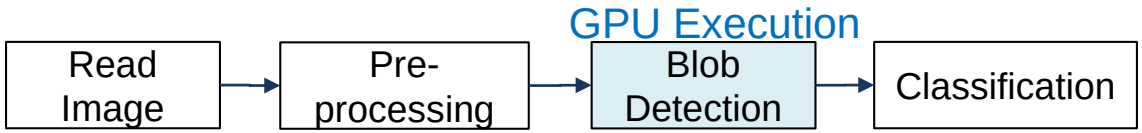
- Motivation & Introduction
- Suspension-based MPCP
- **Evaluation**
  - Case Study
  - Schedulability Experiment
- Conclusions

Case Study

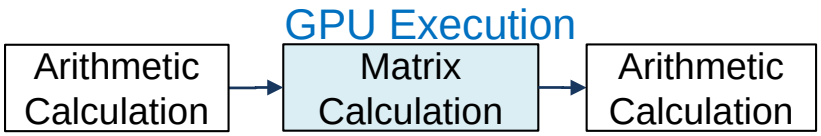
- Motivated by the software system of CMU's self-driving car*
  - **Lane-Change Detector**

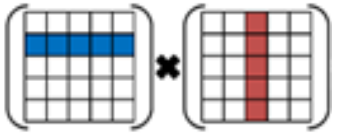


  - **Workzone Detector†**



  - **Matrix Calculation**





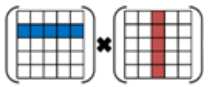
* J. Wei et al. Towards a viable autonomous driving research platform. In IEEE Intelligent Vehicles Symposium (IV), 2013.
† J. Lee et al. Kernel-based traffic sign tracking to improve highway workzone recognition for reliable autonomous driving. In IEEE International Conference on Intelligent Transportation Systems (ITSC), 2013.

Experimental Setup

- NVIDIA TX2
  - ✓ 6 CPU Cores
  - ✓ 1 GPU (256 Cores, Pascal Arch.)

CPU core **1**

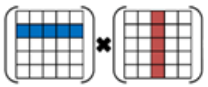
Lane Change Detector



Two Matrix Calculations

CPU core **2**

Workzone Detector



One Matrix Calculation

Task priorities are assigned based on the rate-monotonic policy

DEMO

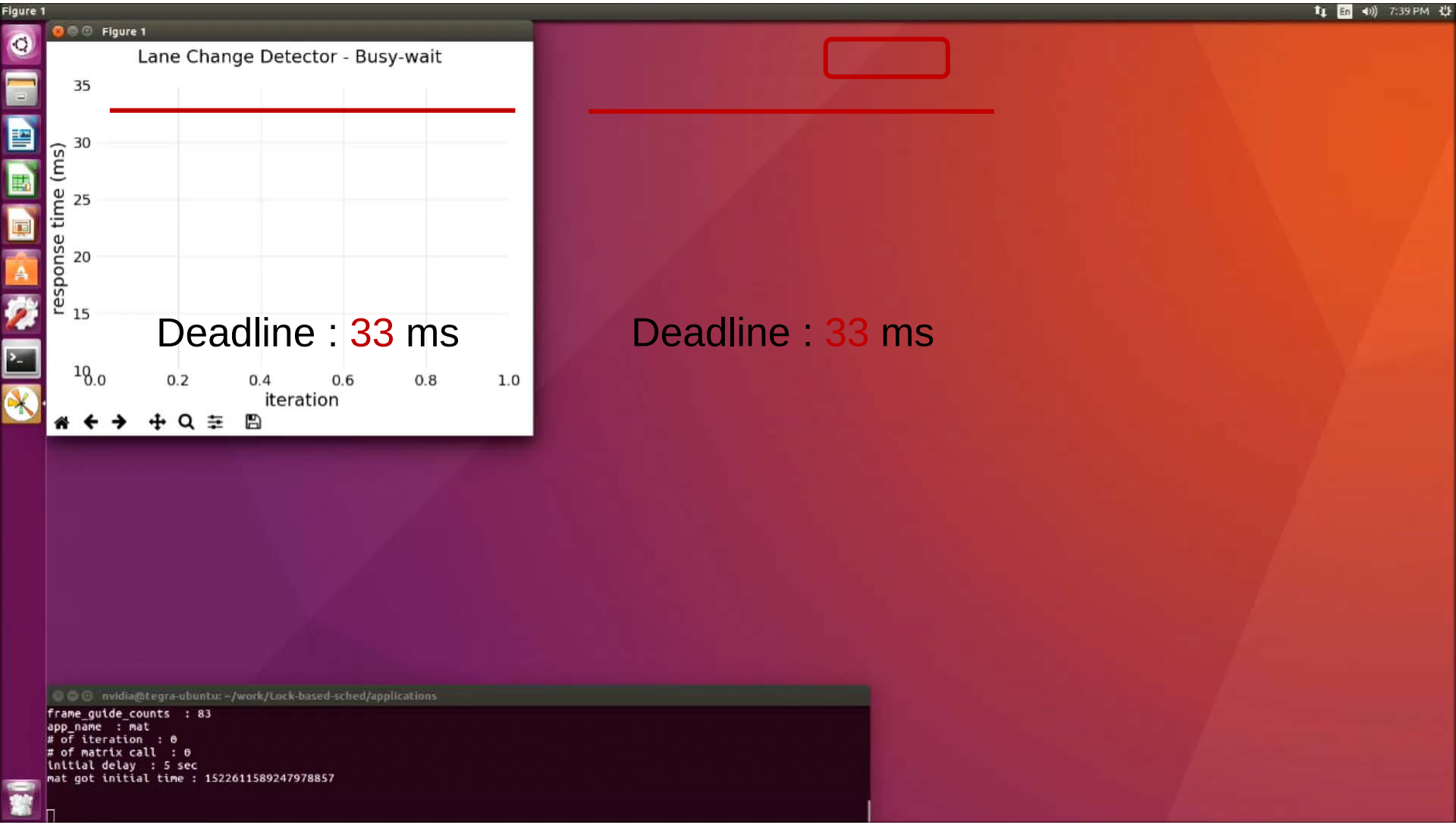
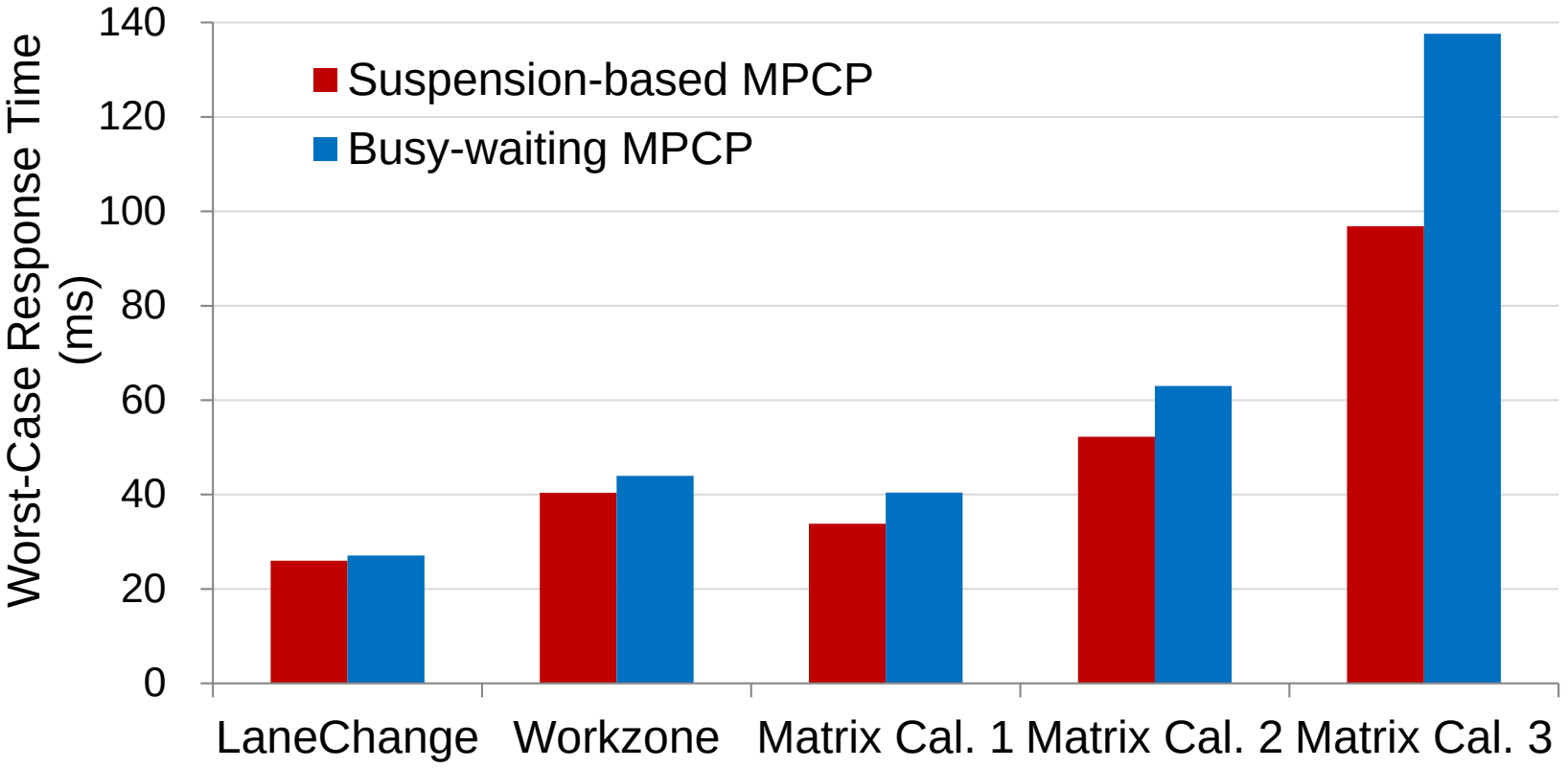


Figure 1
7:39 PM
Figure 1
Lane Change Detector - Busy-wait
response time (ms)
35
30
25
20
15
10
0.0
0.2
0.4
0.6
0.8
1.0
iteration
Deadline : 33 ms
Deadline : 33 ms
nvidia@tegra-ubuntu: ~/work/Lock-based-sched/applications
frame_guide_counts : 83
app_name : mat
# of iteration : 0
# of matrix call : 0
initial delay : 5 sec
mat got initial time : 1522611589247978857

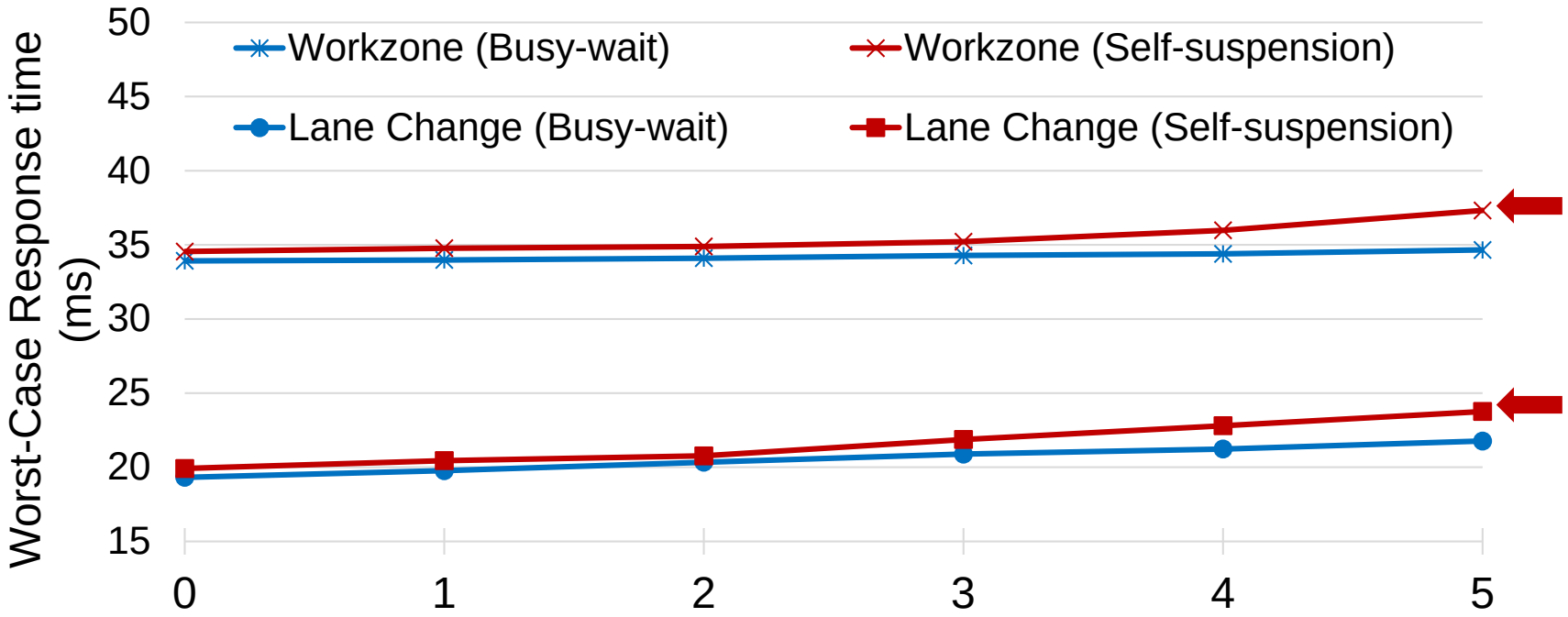
Suspension-based vs. Busy-waiting MPCP



- ✓ Performs better in practice, especially for lower-priority tasks
- ✓ Allows other tasks to use the CPU while a task is using the GPU

Effect of Suspension Overhead

- Test result w.r.t. the number of co-scheduled tasks w/ **small** GPU segments



≈ 200 µs

The **overhead** of self-suspension implementation negatively affects task response times when the tasks have small GPU segments

Schedulability Experiments

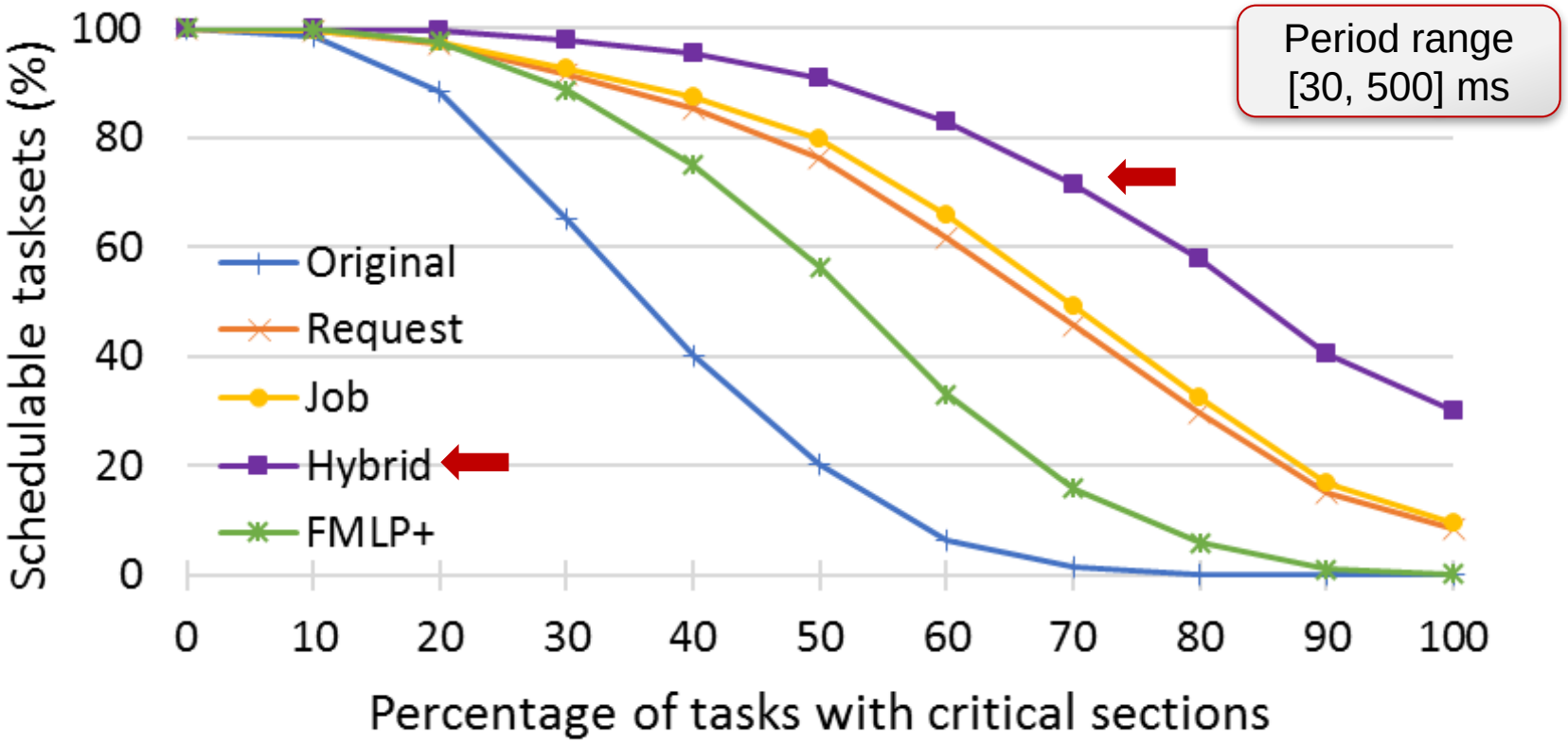
- Purpose: To explore the impact of the different approaches on task schedulability
- 10,000 randomly-generated tasksets

MPCP – Our Analysis
FMLP+ – LP-based Analysis*

Parameters	Values
Number of CPUs (m)	4
Number of shared resources (g)	[1, 3]
Number of tasks per CPU	[3, 6]
Percentage of tasks with critical sections	[10, 40] %
Task period and deadline ($T_i = D_i$)	[30, 500] ms
Utilization per CPU	[40, 60] %
Ratio of crit. Sec. len. To non-crit. Sec. len. (G_i / C_i)	[10, 30] %
Number of critical sections per task (η_i)	[1, 3]
Number of suspensions in a critical section ($\zeta_{i,j}$)	[1, 2]

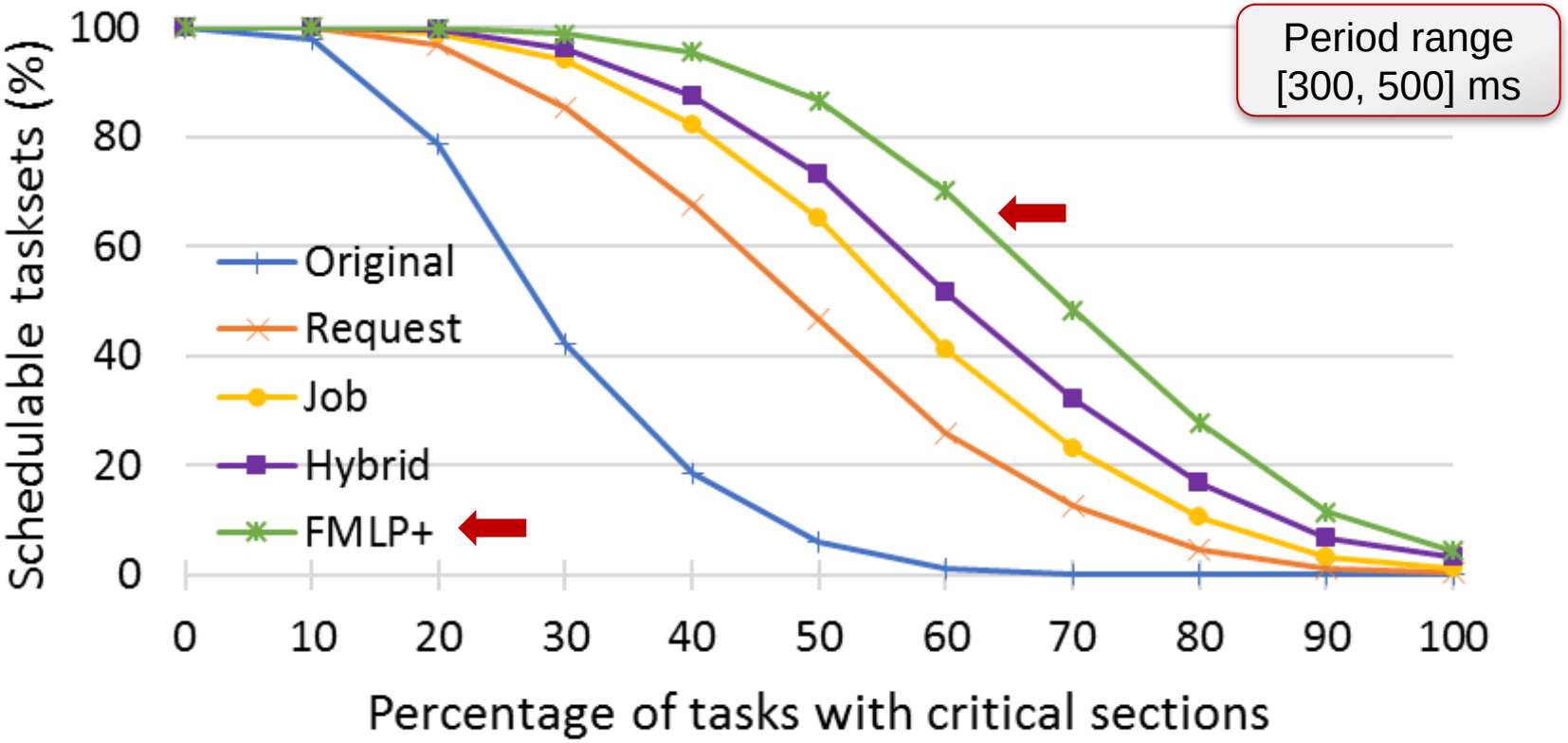
* The Schedulability Test Collection and Toolkit (SchedCAT). http://github.com/brandenburg/schedcat

Schedulability w.r.t. the Percentage of GPU-using Tasks



Hybrid MPCP outperforms both the original MPCP and LP-based FMLP+

Schedulability w.r.t. the Percentage of GPU-using Tasks



Neither hybrid MPCP nor LP-based FMLP+ dominates the other

Hybrid MPCP analysis is over 100x faster than LP-based FMLP+

Conclusions

- Suspension-based MPCP
  - ✓ Motivated by the limitations of the busy-waiting synchronization-based approach
  - ✓ Implementation on a real-world embedded platform
  - ✓ Significant improvement over the busy-waiting approach
  - ✓ Very competitive with and often outperforms LP-based FMLP+
  - ✓ 100x better runtime performance compared to LP-based analysis

- Future directions
  - ✓ A detailed study of the suspension overhead trade-offs on modern platforms with accelerators
  - ✓ Comparison with other synchronization protocols

Thank You

Analytical Enhancements and Practical Insights for MPCP with Self-Suspensions

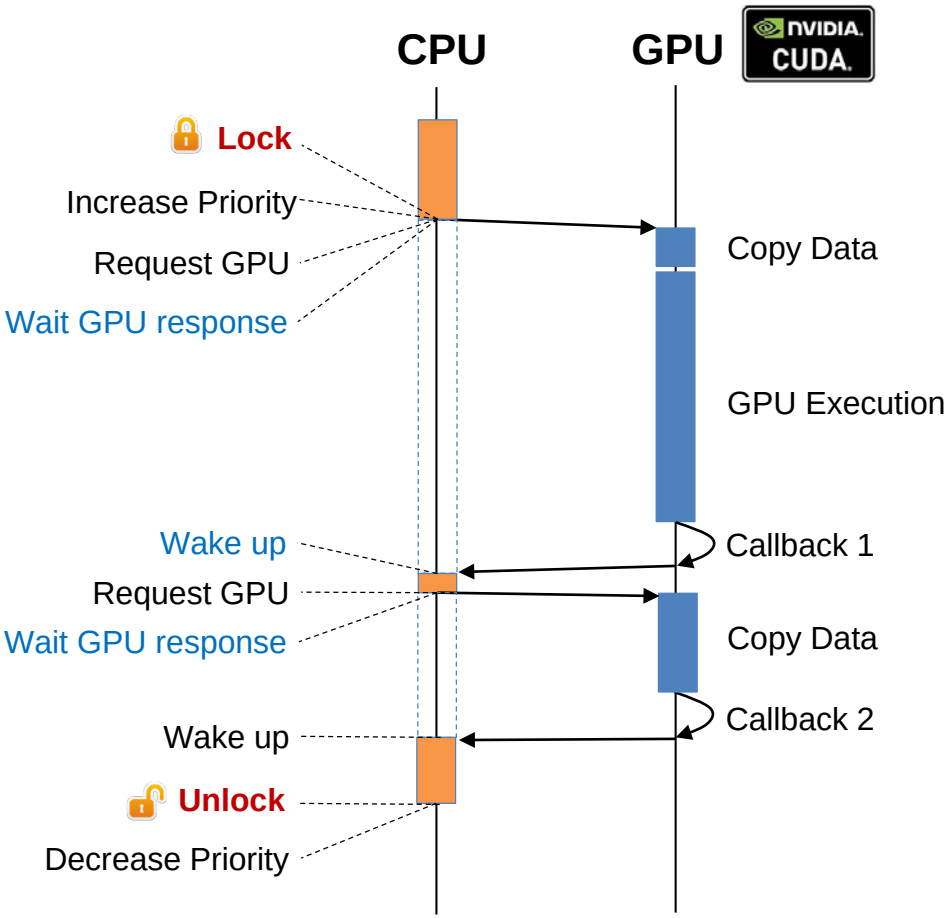
Pratyush Patel*, Iljoo Baek*, Hyoseung Kim†, Raj Rajkumar*

ibaek@andrew.cmu.edu

* Carnegie Mellon University
† University of California, Riverside

BACKUP SLIDES

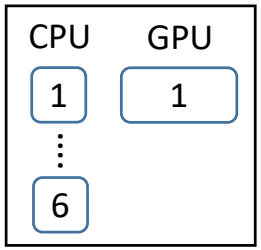
Self-Suspension Implementation



- Global Lock
  - ✓ POSIX **pthread_mutex()**
  - ✓ Shared memory
- Priority Ceiling
  - ✓ sched_setscheduler()
- CPU Suspension
  - ✓ POSIX **pthread_cond()**
- GPU Execution
  - ✓ **Asynchronous** functions
    ex) cudaMemcpyAsync()
  - ✓ Stream and **Callback**

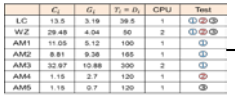
Experimental Setup

- NVIDIA TX2
  - 4 ARM Cortex-A9 at 2GHz
  - 2 Denver at 2GHz
  - 1 GPU (256 Cores, Pascal Arch.)
  - Ubuntu 16.04



LC

WZ



AM1~5

	C_i	G_i	$T_i = D_i$	CPU	Test
LC	13.5	3.19	39.5	1	①②③
WZ	29.48	4.04	50	2	①②③
AM1	11.05	5.12	100	1	①
AM2	8.81	9.38	165	1	①
AM3	32.97	10.88	300	2	①
AM4	1.15	2.7	120	1	②
AM5	1.15	0.7	120	1	③

① **General Use Case**

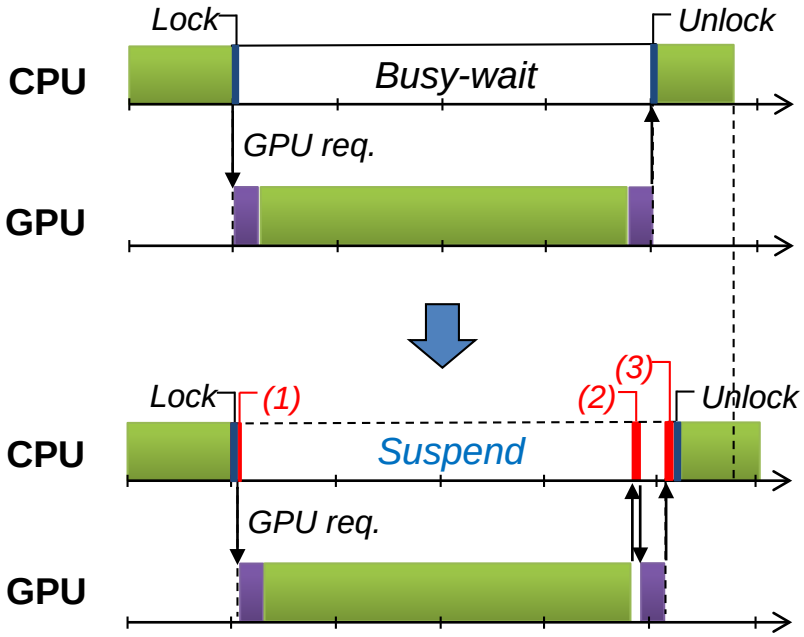
② **GPU Overload**

③ **Overhead Test**

* All times are in milliseconds (ms)

Suspension Overhead

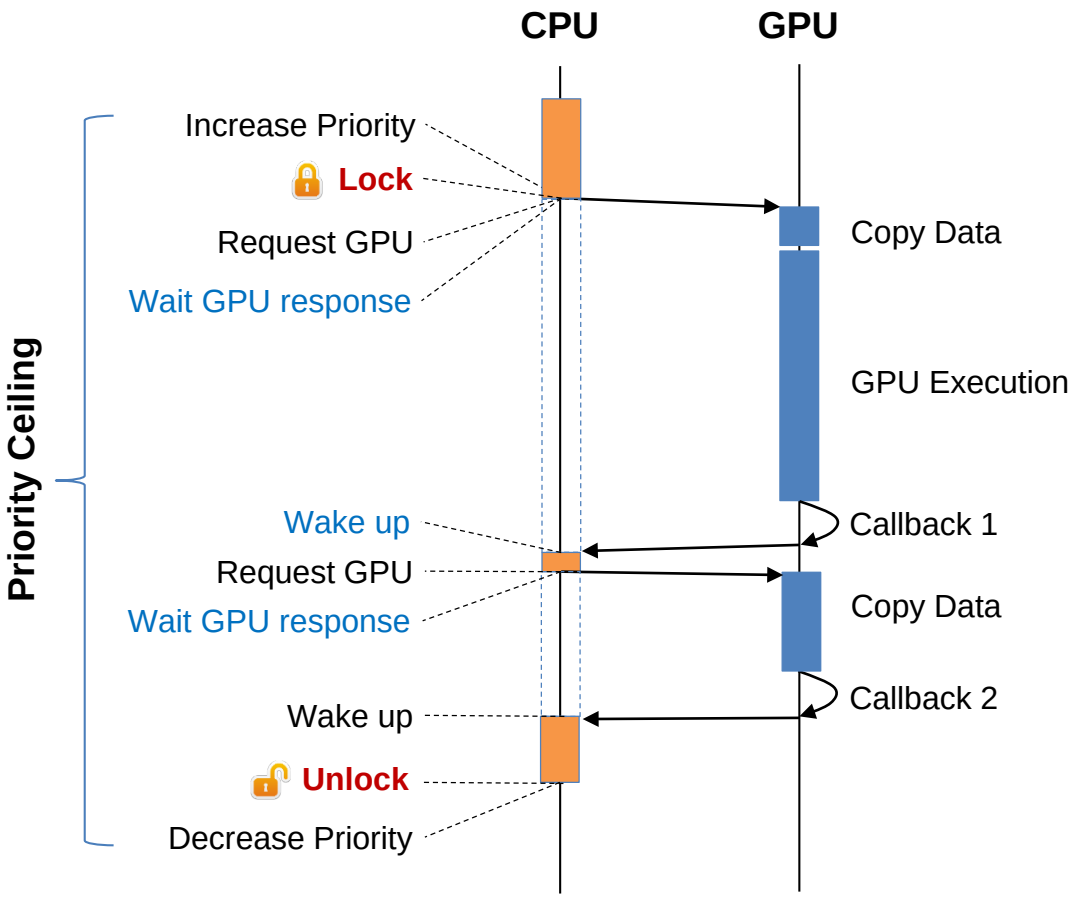
- CPU-side overhead
  - ✓ Suspension
  - ✓ Context Switching
- GPU-side overhead
  - ✓ Asynchronous Calls
  - ✓ Callback functions



≈ 200 μs

GPU access time < the suspension overhead
➔ the busy-wait approach is better

Self-Suspension Implementation



- Global Lock
  - ✓ Shared memory
  - ✓ pthread_mutex()
- Suspension
  - ✓ a POSIX conditional variable
- CUDA-related
  - ✓ Asynchronous CUDA functions
    ex) cudaMemcpyAsync()
  - ✓ Stream and Callback
- Priority Ceiling
  - ✓ sched_setscheduler()

```
new_pri =
  highest_pri + (cur_pri – lowest_pri) + 1

Ex)
      Task 1: 80 (cpu)
      Task 2: 54 (gpu) -> 85 = 80 + (54 - 50) + 1
      Task 3: 53 (gpu) -> 84 = 80 + (53 - 50) + 1
      Task 4: 52 (gpu) -> 83 = 80 + (52 - 50) + 1
      Task 6: 50 (gpu) -> 81 = 80 + (50 - 50) + 1
```

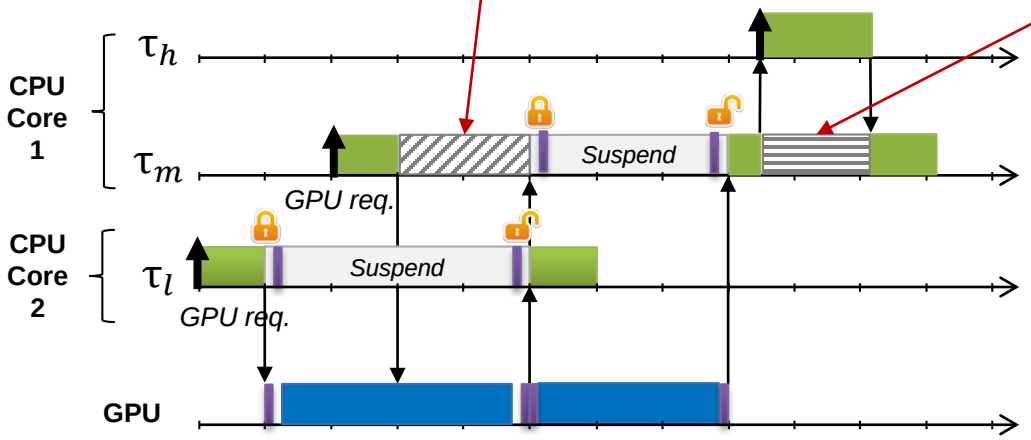
Task Response Time with Suspension

- Worst-Case Response Time (W_i)
  - Time span between a request and the end of the request

$$W_i^{n+1} = C_i + G_i + B_i + \sum_{\tau_h \in hpp(\tau_i)} \left\lceil \frac{W_i^n + W_h - C_h}{T_h} \right\rceil \cdot C_h$$

Blocking Delay

Preemption Delay by Higher priority task



$W_i \leq$ Deadline (D_i)

= Schedulable

Task arrival | CPU execution | GPU execution | Blocked segment | Preempted segment

Schedulability Experiments

- Purpose: To explore the impact of the different approaches on task schedulability
- Schedulability: How many taskets are schedulable?

$(C_i, G_i, D_i, T_i, \eta_i)$

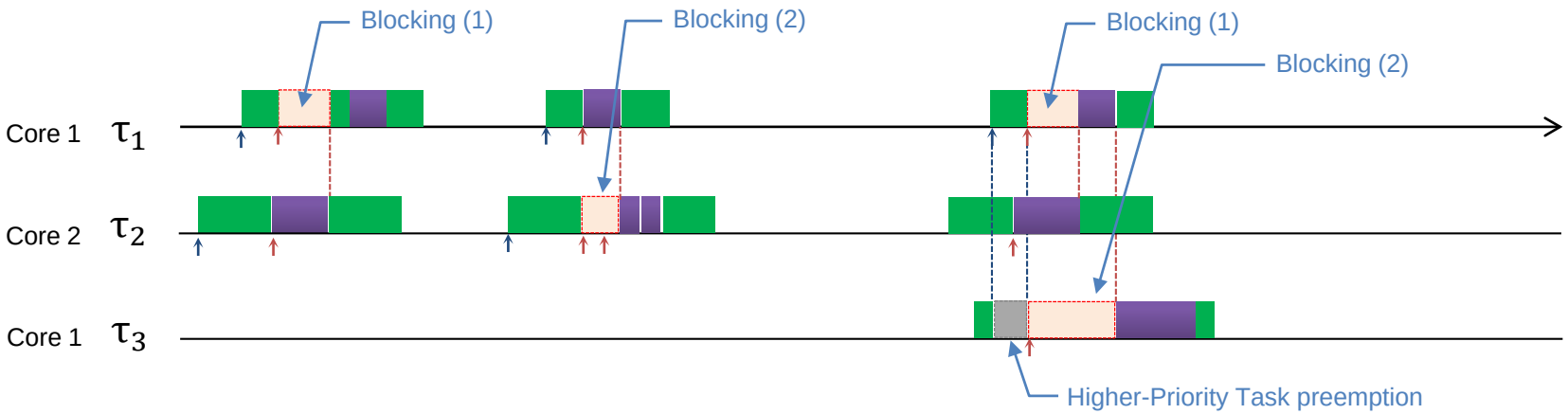
Taskset 1 = $\{\tau_1, \tau_2 \ldots . \tau_x\}$

⋮

Taskset n = $\{\tau_1, \tau_2 \ldots . \tau_x\}$

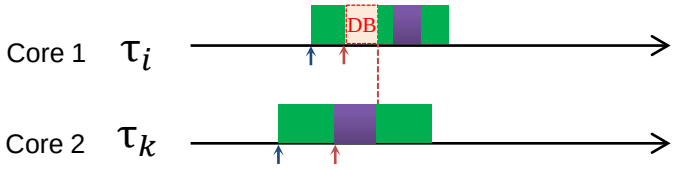
Blocking Definition

- A task τ_i is said to be blocked
  - ✓ If a local task τ_i with a lower base priority is scheduled while of τ_i is pending.
  - ✓ If any task τ_k has locked the resource that τ_i is waiting for.

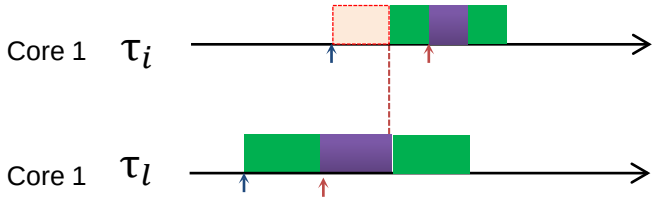


Blocking Definition

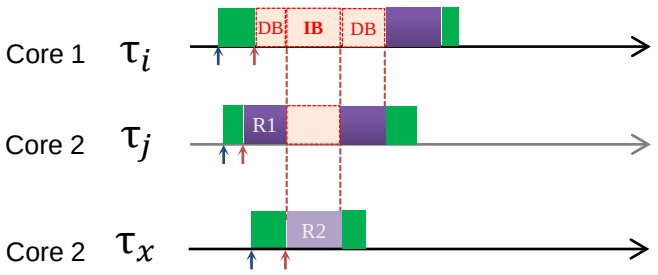
- **Direct Blocking (DB)**
  - is incurred when any task τ_k has locked the resource that τ_i is waiting for.



- **Prioritized Blocking (PB)**
  - is incurred when lower-priority tasks executing with priority ceilings preempt the CPU execution of τ_i



- **Indirect Blocking (IB)**
  - is incurred when a task τ_x accessing a resource with a higher priority ceiling preempts the execution of τ_j , which is holding the resource that τ_i is waiting for.



Task Response Time with Suspension

- **Worst-case response time** under partition fixed-priority scheduling

$$W_i = C_i + G_i + B_i + \sum_{\tau_h \in hpp(\tau_i)} I_i(\tau_i)$$

Worst-case execution time of τ_i

Upper bound on the maximum synchronization-based blocking

Worst-case preemption time due to higher priority tasks on the same CPU

$$I_i(\tau_i) = \alpha_{i,h} \cdot E_h$$

Worst-case CPU execution time of τ_i

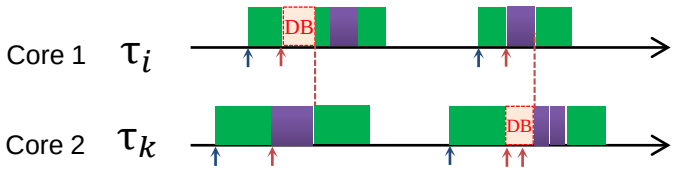
$$\alpha_{i,h} = \left\lceil \frac{W_i + W_h - E_h}{T_h} \right\rceil$$

Back-to-back execution effect

Maximum instances of τ_h released during the execution of τ_i

Blocking Analysis

- **Direct Blocking (DB)**
  - is incurred when any task τ_k has locked the resource that τ_i is waiting for.



✓ **Request-Driven (RD) Approach**

$$B_i^{dr} = \sum_{0<j\leq\eta_i} B_{i,j}^{dr}$$

Critical sections per job of τ_i

Worst-case response time of k[th] critical section of τ_l

$$B_{i,j}^{dr} = \max_{\substack{\tau_l\in lp(\tau_i) \\ \wedge R(\tau_{i,j})=R(\tau_{l,k})}} (H_{l,k}) + \sum_{\substack{\tau_h\in hp(\tau_i) \\ \wedge R(\tau_{i,j})=R(\tau_{h,k})}} \beta_{i,j,h}\cdot H_{h,k}$$

$$\sum_{\tau_h\in hp(\tau_i)} \sum_{0<k\leq\eta_h} \beta_{i,j,h}\cdot H_{h,k}$$

$$\beta_{i,j,h} = \left\lceil \frac{B_{i,j}^{dr} + W_h - E_h}{T_h} \right\rceil$$

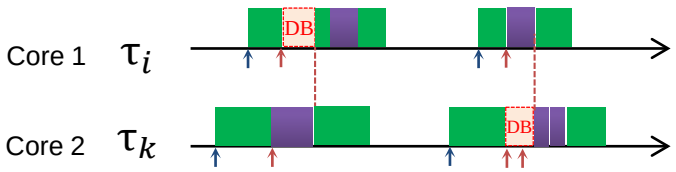
Maximum blocking from lower priority tasks

Maximum blocking from higher priority tasks

Maximum instances of τ_h released during the blocking duration $B_{i,j}^{dr}$

Blocking Analysis

- **Direct Blocking (DB)**
  - is incurred when any task τ_k has locked the resource that τ_i is waiting for.



- ✓ **Job-Driven (JD) Approach**

Maximum number of times τ_i accesses resource R_r

iff. $R(\tau_{l,k}) = R(\tau_i)$

$$B_i^{dj} = \sum_{r \in R(\tau_i)} \eta_{i,r} \cdot \max_{\substack{\tau_l \in lp(\tau_i) \\ \wedge R(\tau_{l,k}) = r}} (H_{l,k}) + \sum_{\substack{\tau_h \in hp(\tau_i) \\ \wedge R(\tau_{h,k}) \in R(\tau_i)}} \alpha_{i,h} \cdot H_{h,k}$$

$$\sum_{0<j\leq\eta_i} \sum_{\tau_l \in lp(\tau_i)} \sum_{0<k\leq\eta_h} \max(H_{l,k})$$

$$\sum_{\tau_h \in hp(\tau_i)} \sum_{0<k\leq\eta_h} \alpha_{i,h} \cdot H_{h,k}$$

iff. $R(\tau_{h,k}) = R(\tau_i)$

$$\alpha_{i,h} = \left\lceil \frac{W_i + W_h - E_h}{T_h} \right\rceil$$

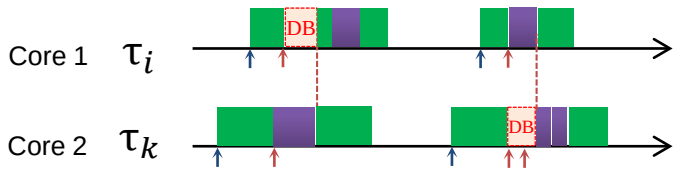
Maximum direct blocking caused by lower-priority tasks
= analogous to the RD approach

Direct blocking caused by each higher-priority task critical section that uses the requested resource

Maximum instances of τ_h released during the execution of τ_i

Blocking Analysis

- **Direct Blocking (DB)**
  - is incurred when any task τ_k has locked the resource that τ_i is waiting for.



✓ **Hybrid Approach**

$$B_i^{dm} = B_i^{dml} + B_i^{dmh}$$

$$\alpha_{i,h} = \left\lceil \frac{W_i + W_h - E_h}{T_h} \right\rceil \quad \beta_{i,j,h} = \left\lceil \frac{B_{i,j}^{dr} + W_h - E_h}{T_h} \right\rceil$$

1) Maximum direct higher-priority blocking

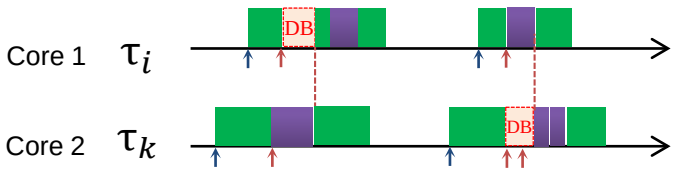
$$B_i^{dmh} = \sum_{\substack{\tau_h \in hp(\tau_i) \\ \wedge R(\tau_{h,k}) = R(\tau_i)}} \delta_{i,h} \cdot H_{h,k}$$

$$\delta_{i,h} = \min\left(\alpha_{i,h}, \sum_{\substack{0 < j \le \eta_i \\ \wedge R(\tau_{h,k}) \in R(\tau_i)}} \beta_{i,j,h} \right)$$

- $H_{h,k}$: Worst-case response time of k^{th} critical section of τ_h
- $\delta_{i,h}$: Maximum cumulative number of interfering requests by τ_h to the resources accessed by critical sections of τ_i
- $\alpha_{i,h}$: Maximum instances of τ_h released during the execution of τ_i
- $\beta_{i,j,h}$: Maximum instances of τ_h released during the blocking duration $B_{i,j}^{dr}$

Blocking Analysis

- **Direct Blocking (DB)**
  - is incurred when any task τ_k has locked the resource that τ_i is waiting for.



✓ **Hybrid Approach**

$$B_i^{dm} = B_i^{dml} + B_i^{dmh}$$

2) Maximum direct lower-priority blocking

$$B_i^{dml} = \sum_{0<j\leq g} \sum_{0<k\leq|Q_{i,j}|} \psi_{i,j,k} \cdot L_{i,j,k}$$

$$\psi_{i,j,k} = \min\left(\eta_{i,j} - \sum_{0<t<k} \psi_{i,j,t} \ , \theta_{i,X_{i,j,k}}\right)$$

$$\theta_{i,l} = \left\lceil \frac{W_i + D_l - E_l}{T_l} \right\rceil$$

$$Q_{i,j} = \{H_{l,x} | \tau_l \in lp(\tau_i) \wedge R(\tau_{l,k}) = r\}$$

The k[th] longest critical section that access resource R_j and belong to $lp(\tau_i)$

Maximum number of times τ_i accesses resource R_j

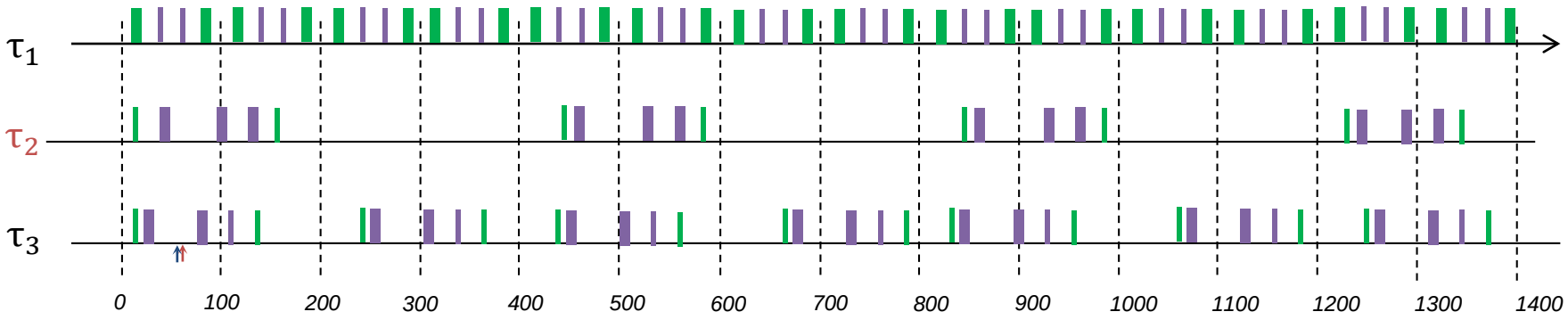
Maximum number of times that the k[th] longest critical section can block τ_i

A set contains the worst-case execution times of all the critical sections that access resource R_j and belong to $lp(\tau_i)$

Maximum number of release of a lower-priority τ_l during the execution of τ_i

Blocking Analysis: Example

Task	C_i	G_i	n_i	$G_{i,1}$	$G_{i,2}$	$G_{i,3}$	T
τ_1	20	10	2	5	5	-	100
τ_2	10	30	3	10	10	10	400
τ_3	10	25	3	10	10	5	200



$$Q_{2,1} = \{H_{l,x} | \tau_l \in lp(\tau_2) \wedge R(\tau_{l,k}) = 1\} = 20, 10, 5$$

$$L_{2,1,1} = 20 \quad L_{2,1,2} = 10 \quad L_{2,1,3} = 5$$

$$\theta_{2,3} = 1$$

$$\psi_{2,1,1} = \min\left(\eta_{2,1} - \sum_{0<t<1} \psi_{2,1,t}\ , \theta_{2,3}\right) = \min(3, 1) = 3$$

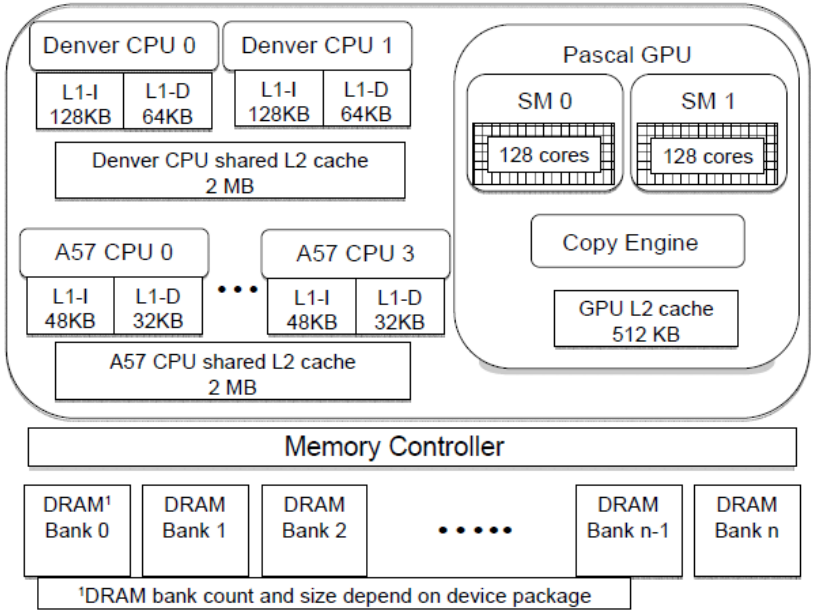
$$\psi_{2,1,2} = \min\left(\eta_{2,1} - \sum_{0<t<2} \psi_{2,1,t}\ , \theta_{2,3}\right) = \min(0, 1) = 0$$

Nvidia TX2

Nvidia TX2

	NVIDIA Jetson TX1	NVIDIA Jetson TX2
CPU	ARM Cortex-A57 (quad-core) @ 1.73GHz	ARM Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz
GPU	256-core Maxwell @ 998MHz	256-core Pascal @ 1300MHz
Memory	4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s	8GB 128-bit LPDDR4 @ 1866Mhz | 59.7 GB/s
Storage	16GB eMMC 5.1	32GB eMMC 5.1
Encoder*	4Kp30, (2x) 1080p60	4Kp60, (3x) 4Kp30, (8x) 1080p30
Decoder*	4Kp60, (4x) 1080p60	(2x) 4Kp60
Camera†	12 lanes MIPI CSI-2 | 1.5 Gb/s per lane | 1400 megapixels/sec ISP	12 lanes MIPI CSI-2 | 2.5 Gb/sec per lane | 1400 megapixels/sec ISP
Display	2x HDMI 2.0 / DP 1.2 / eDP 1.2 | 2x MIPI DSI	
Wireless	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.0	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.1
Ethernet	10/100/1000 BASE-T Ethernet	
USB	USB 3.0 + USB 2.0	
PCIe	Gen 2 | 1×4 + 1 x1	Gen 2 | 1×4 + 1×1 or 2×1 + 1×2
CAN	Not supported	Dual CAN bus controller
Misc I/O	UART, SPI, I2C, I2S, GPIOs	
Socket	400-pin Samtec board-to-board connector, 50x87mm	
Thermals‡	-25°C to 80°C	
Power††	10W	7.5W
Price	$299 at 1K units	$399 at 1K units

Nvidia TX2



- SM is limited to 2,048 per SM.
- Shared memory usage is limited to 64KB per SM and 48KB per block.
- the total number of threads each block can use is limited to 1,024.
- Each thread can use up to 255 registers
- A block can use up to 32,768 registers (regardless of its thread count).
- Additionally, there is a limit of 65,536 registers in total on each SM.