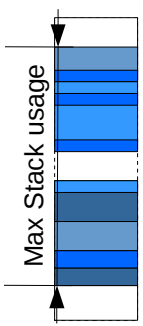
EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing

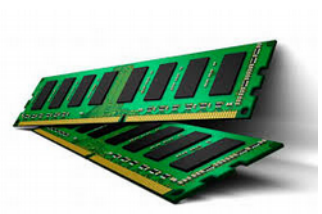
Sebastian Altmeyer, Reinder J. Bril, Paolo Gai

Reconciling **Predictability** and **Performance**

RTOS Stack Implementation:

- enables precise **timing verification**
- enables **optimization of memory/cache layout**





- reduces **stack usage**
- limits **runtime overheads**

Content

1) **System Assumptions**
2) Current Stack Implementations
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions

System Assumptions

- **single processor** system
- **direct address-mapping** memory to cache (no MMU)
- n **tasks:** $\tau_1, \tau_2, ..., \tau_n$ (precedence constraints allowed)
- **priority driven scheduling** (FPPS, FPTS, EDF…)
- **early blocking** resource access protocol
- ***no suspension****/no data left on the stack between task instances*
- *maximum* ***stack usage*** SU_i *per task* τ_i

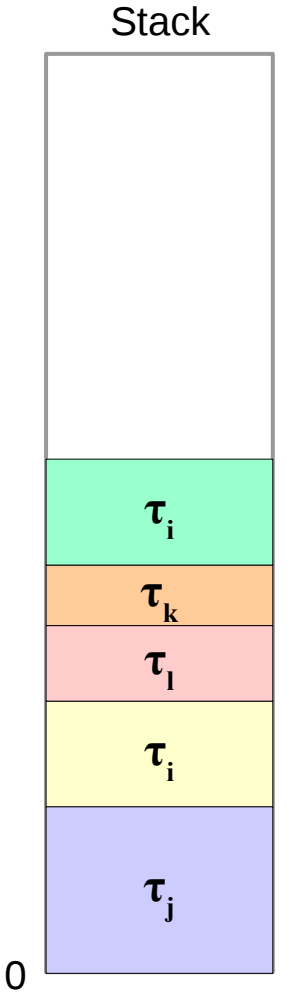
Content

1) System Assumption
2) **Current Stack Implementations**
3) EMPRESS
4) Evaluation/Case Study
5) Conclusions

Dedicated Stacks

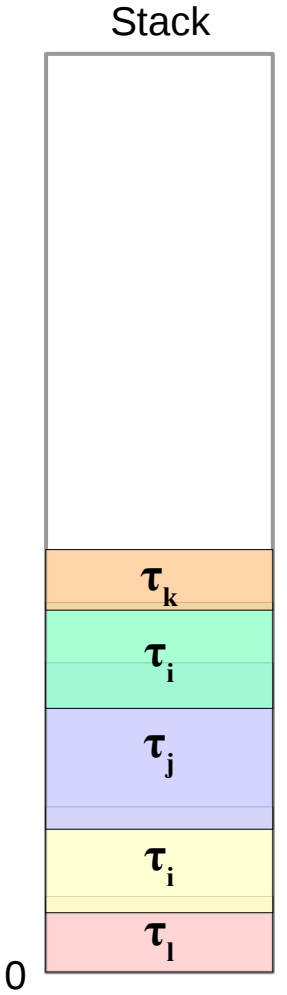
- each task is given **dedicated stack area**
- **unique stack pointer** per task
- total stack usage: $\sum SU_i$

  ➔ potentially wasted memory

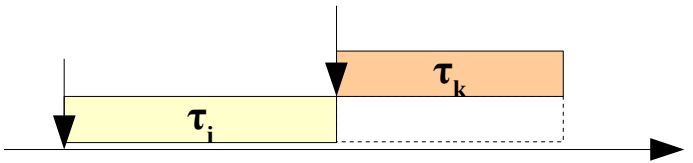


Shared Stack

- all **tasks share stack area**
- **variable stack pointer** per task
  - ➔ potentially reduced stack usage

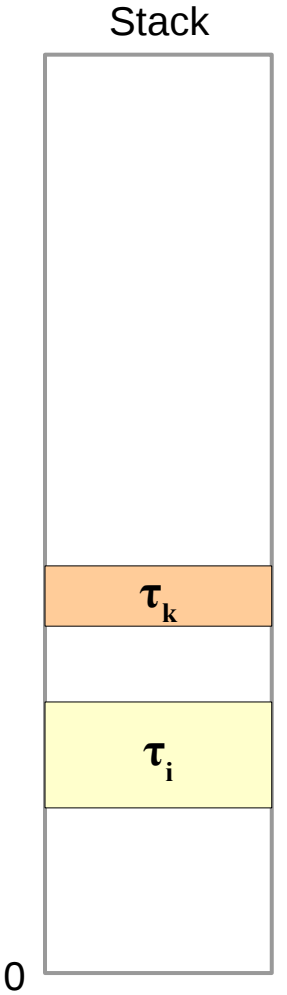


Dedicated Stacks



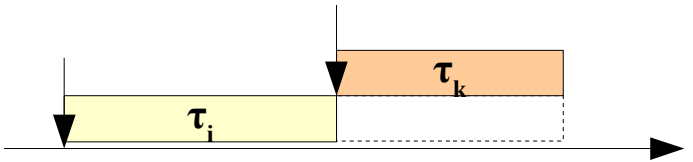
possible schedule

- Stack pointer **independent of preemption point/scenario**
- Stack pointer **statically known**
- potentially **memory gaps** during runtime

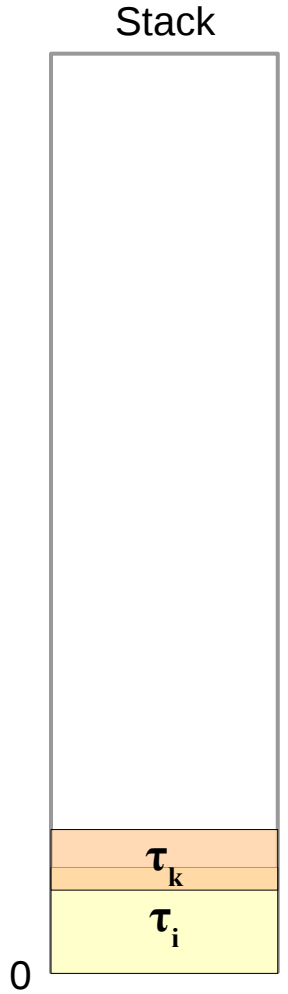


corresponding stack layout

Shared Stack



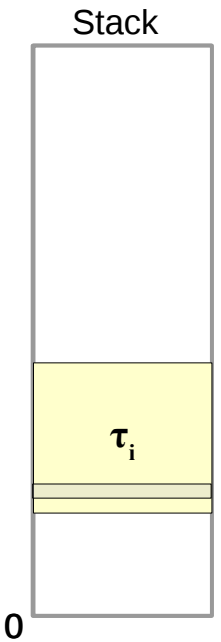
possible schedule



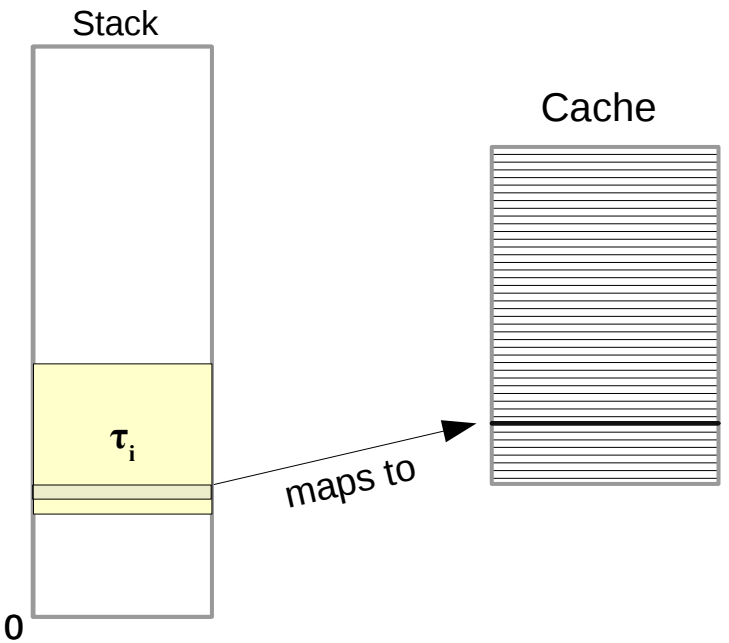
corresponding stack layout

- Stack pointer **depend on preemption point/scenario**
- Stack pointer **dynamically computed**
- no memory gaps

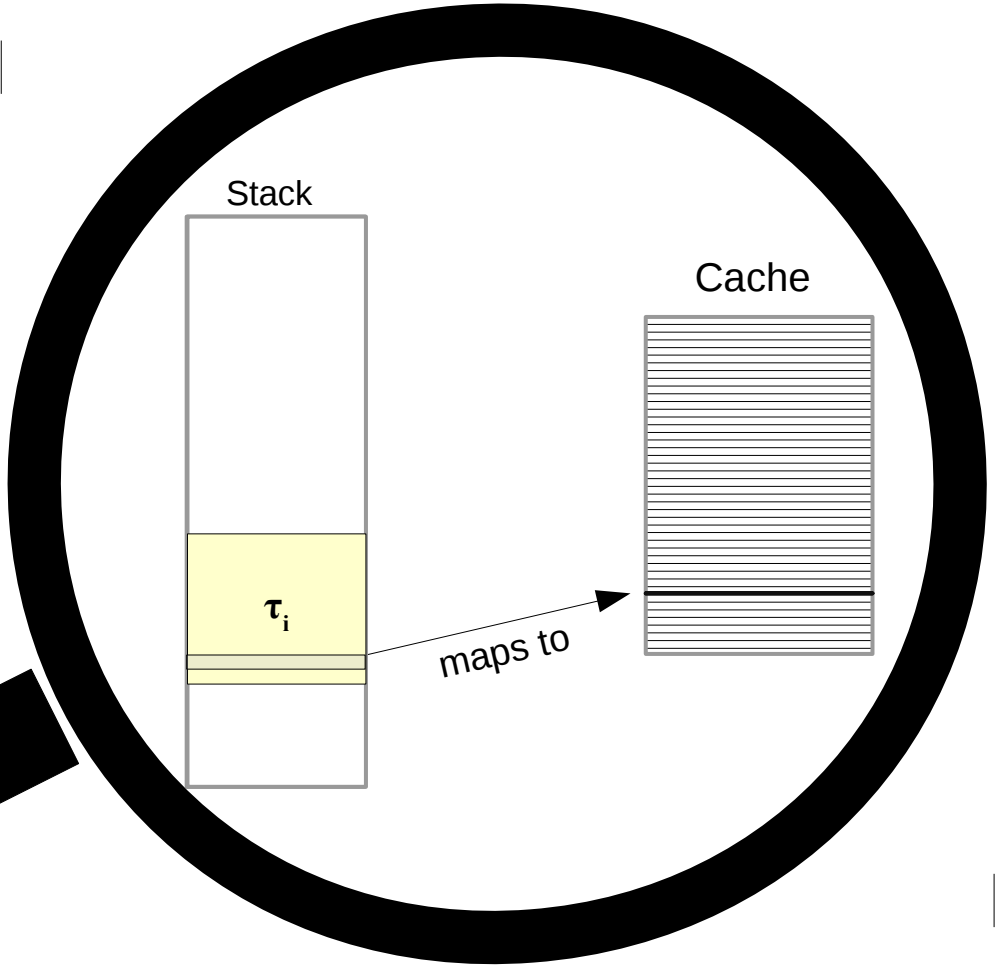
Timing Analysis and **Static** Stack Pointer



Timing Analysis and **Static** Stack Pointer

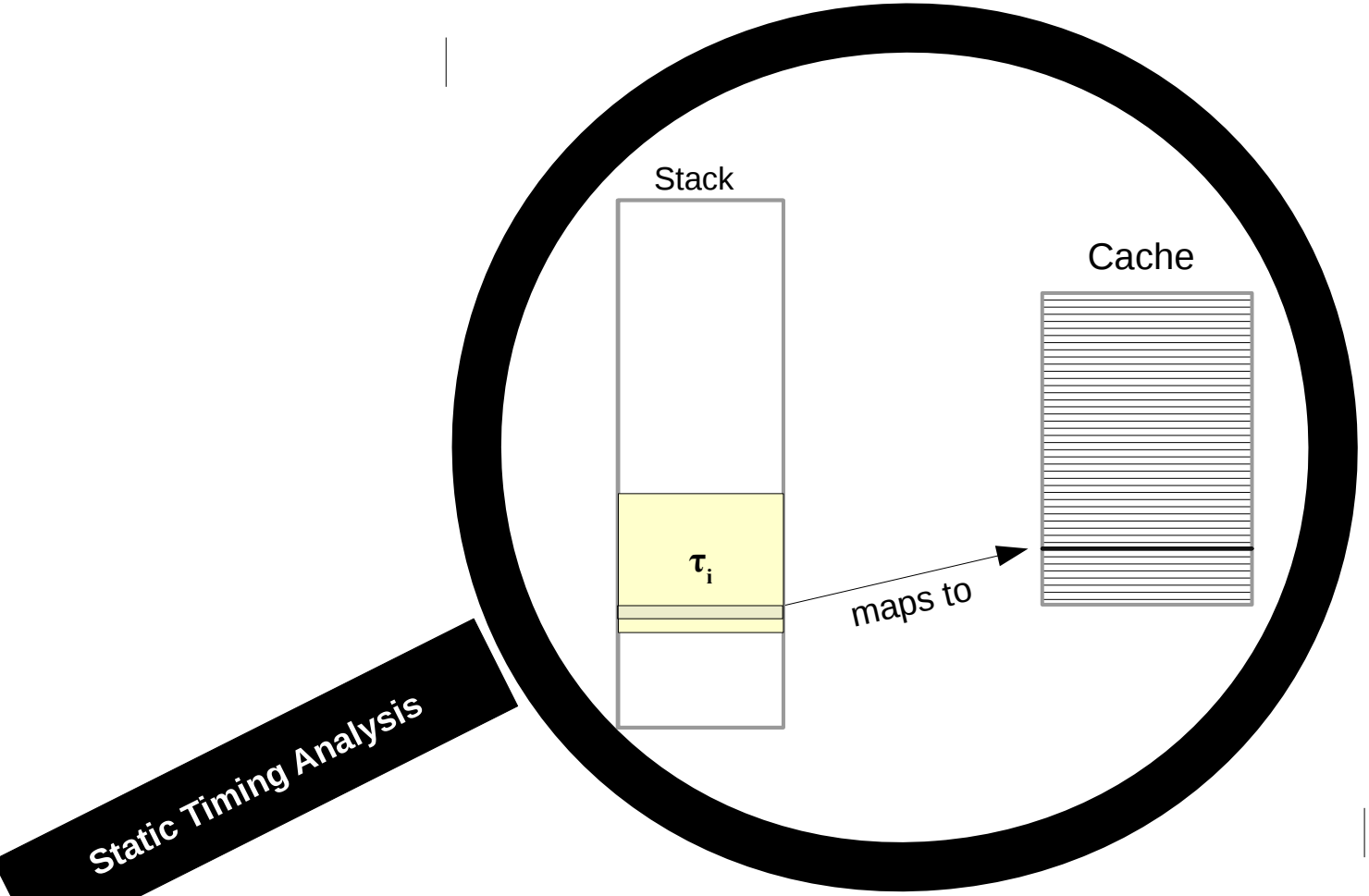


Timing Analysis and **Static** Stack Pointer



Static Timing Analysis

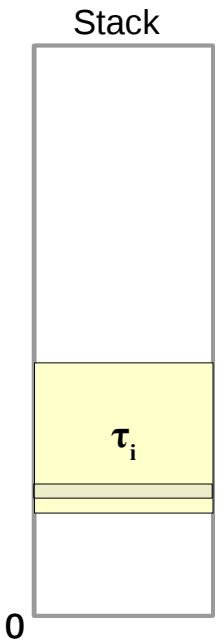
Timing Analysis and **Static** Stack Pointer



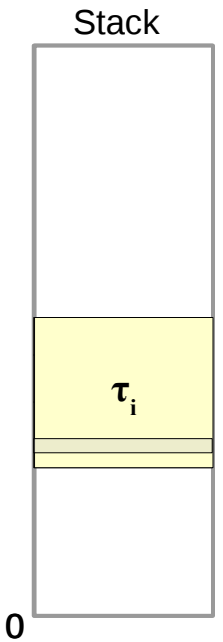
Classify

- Cache Hit
- Cache Miss

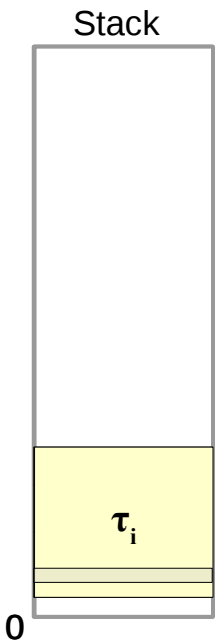
Timing Analysis and **Variable** Stack Pointer



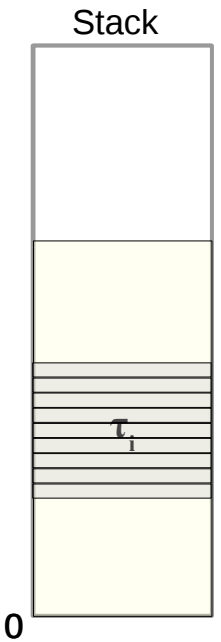
Timing Analysis and **Variable** Stack Pointer



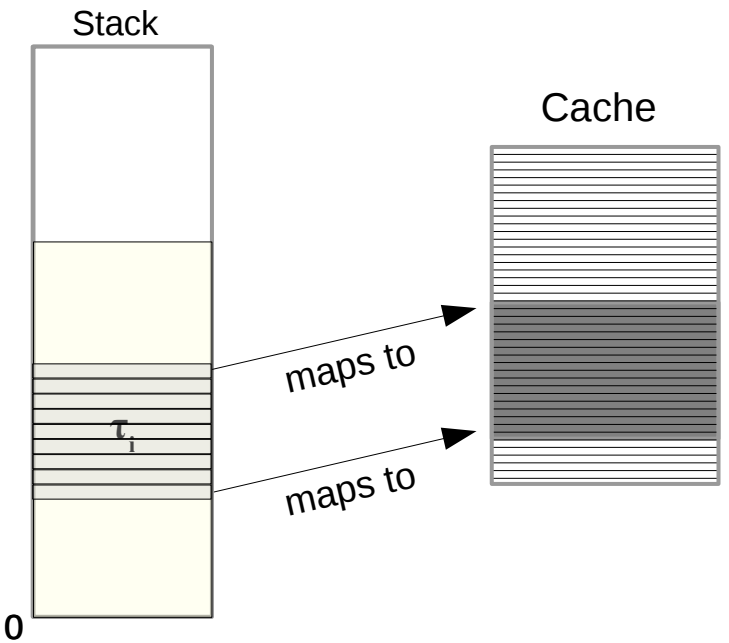
Timing Analysis and **Variable** Stack Pointer



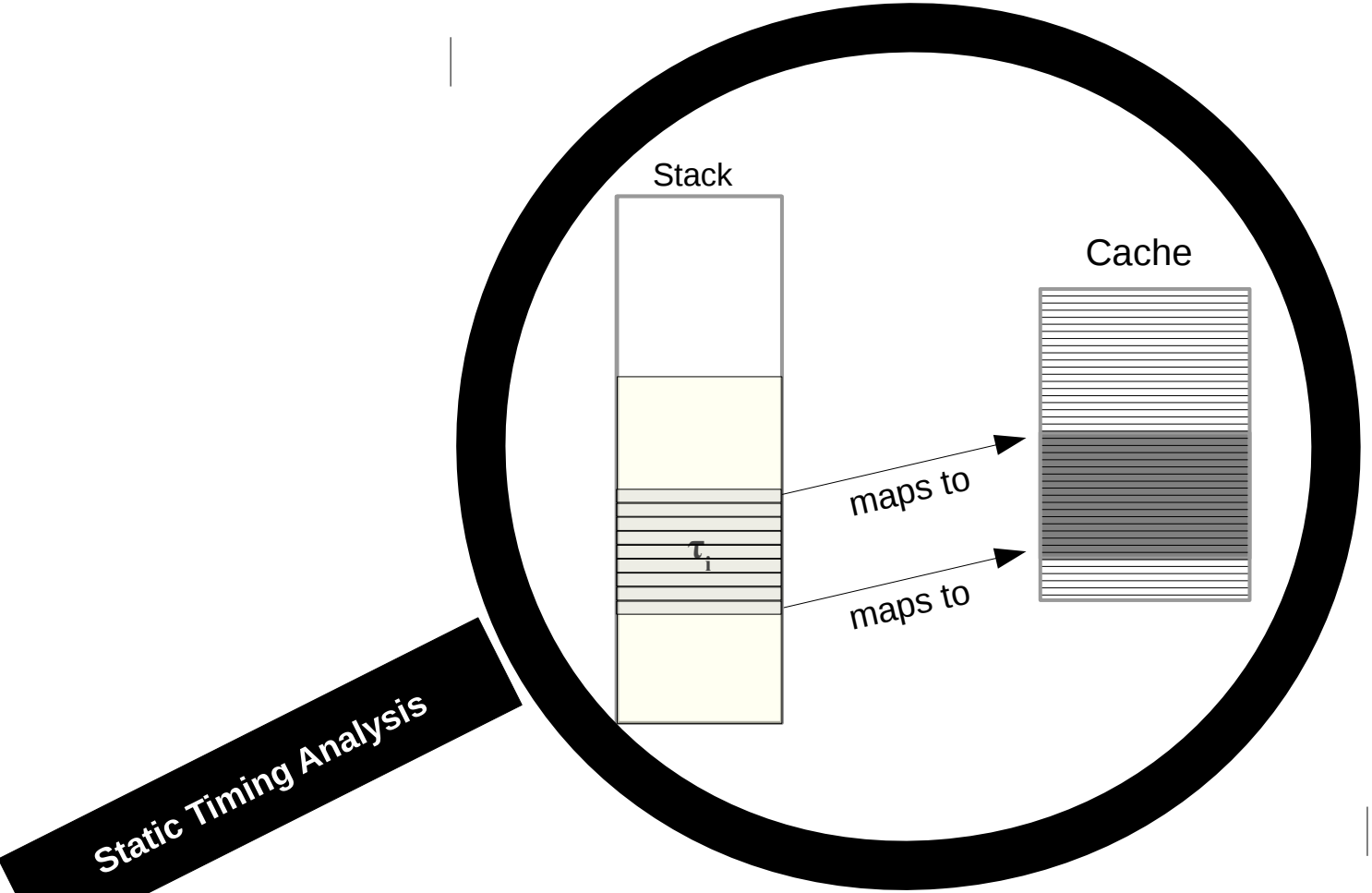
Timing Analysis and **Variable** Stack Pointer



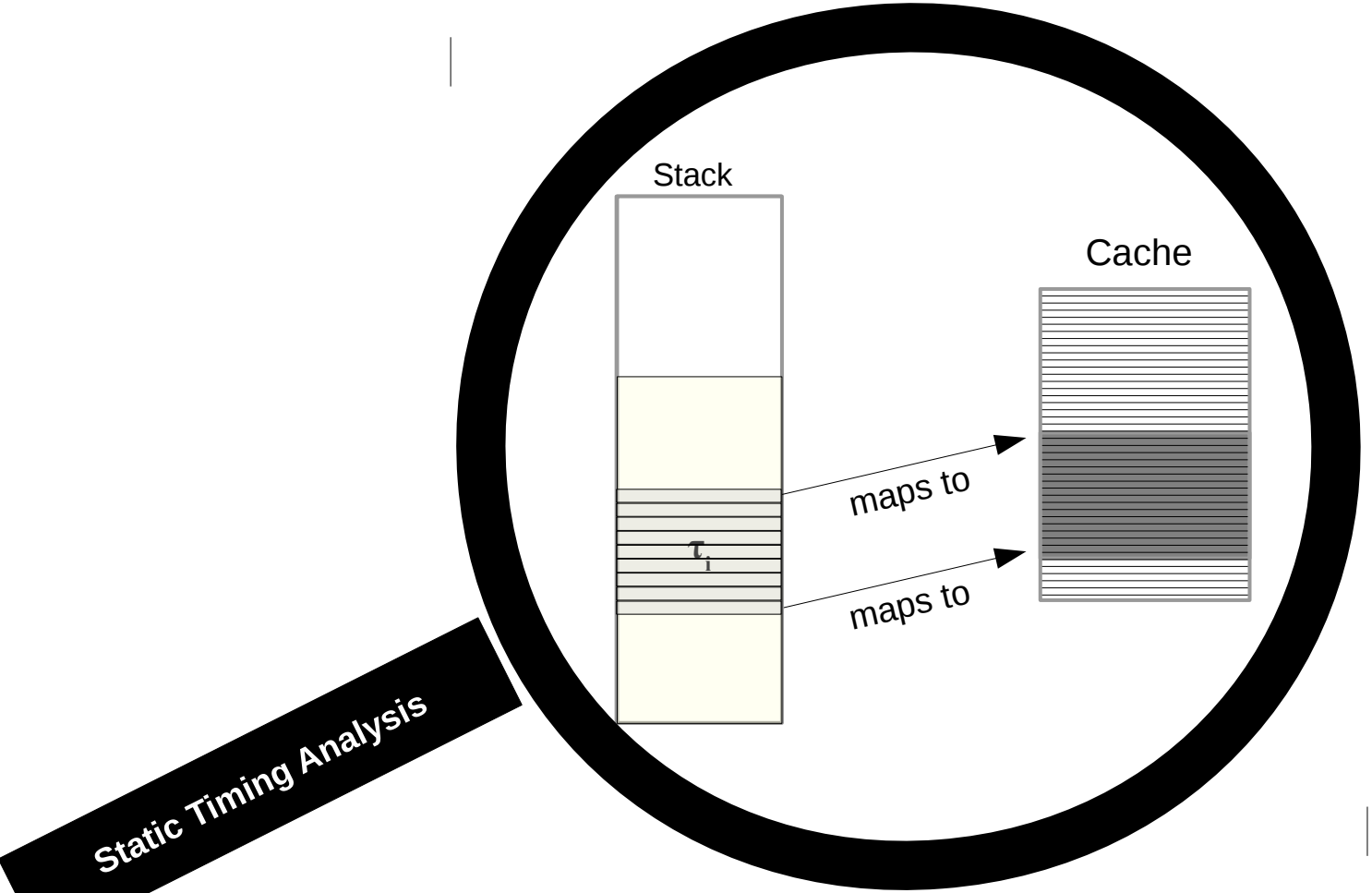
Timing Analysis and **Variable** Stack Pointer



Timing Analysis and **Variable** Stack Pointer



Timing Analysis and **Variable** Stack Pointer



Classify

- ~~Cache Hit~~
- Cache Miss

Timing Analysis and **Variable** Stack Pointer

Access to an unknown address:

- Assumed a **cache miss**
- **Pollutes** abstract cache state
- Results in unknown **access times** (NUMA Architecture)
- **Prevents optimization** of the cache/memory layout

Absint's aiT Timing analyzer guesses a stack pointer
if none is provided [12]

Static timing analysis implies dedicated stacks

Content

1) System Assumptions
2) Current Stack Implementation
3) **EMPRESS**
4) Evaluation/Case Study
5) Conclusions

EMPRESS: an **E**fficient and effective **M**ethod for **PRE**dictable **S**tack **S**haring

Idea:

Static Stack Pointer = Worst-Case Address in Shared Stack

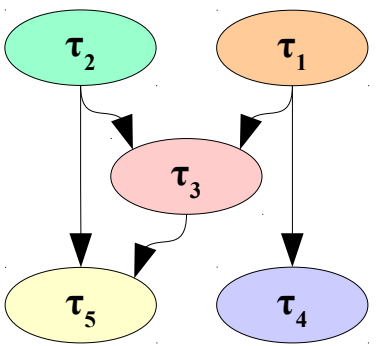
- Static stack pointer
- Stack sharing of mutually non-preemptive tasks
- Stack usage of shared stack under worst-case assumptions

Building the Preemption Graph

- Preemption graph

using:

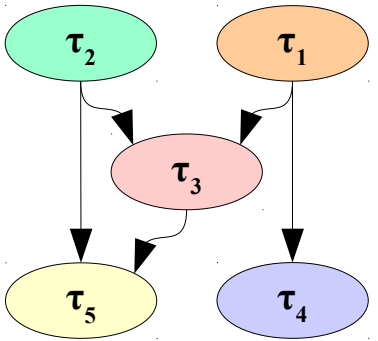
- priorities
- periods/deadlines
- precedence constraints
- etc.



Computing the Stack Addresses

Select worst-case stack pointer **SA** based on

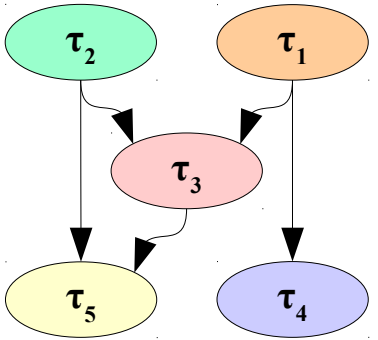
- preemption graph
- SU = max Stack usage of a task



Computing the Stack Addresses

Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



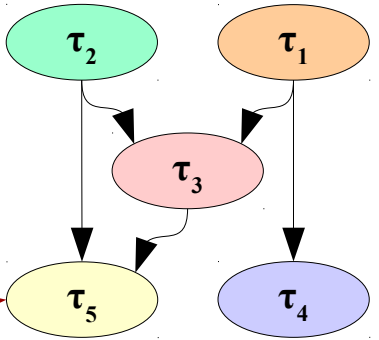
ALGORITHM

for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
$\mathbf{SA_i = 0}$ // set initial stack address to 0
for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
if $\boldsymbol{\tau_i}$ **can preempt** $\boldsymbol{\tau_j}$ *then*
$\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$

Computing the Stack Addresses

Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



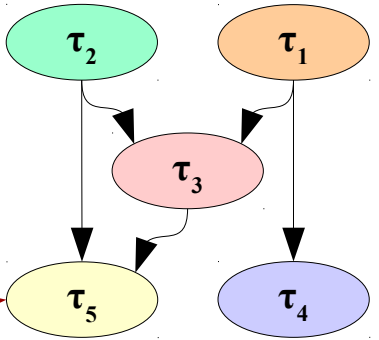
ALGORITHM

for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
  $\mathbf{SA_i = 0}$ // set initial stack address to 0
  for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
    if $\boldsymbol{\tau_i}$ **can preempt** $\boldsymbol{\tau_j}$ *then*
      $\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$

Computing the Stack Addresses

Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



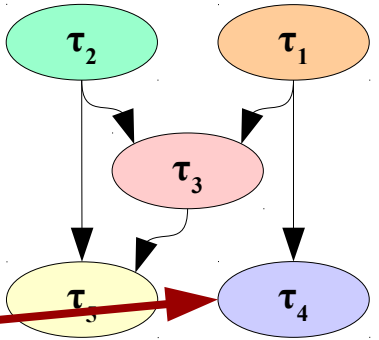
ALGORITHM

for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
  $\mathbf{SA_i = 0}$ // set initial stack address to 0
  for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
    if τ_i **can preempt** τ_j *then*
      $\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$

Computing the Stack Addresses

Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



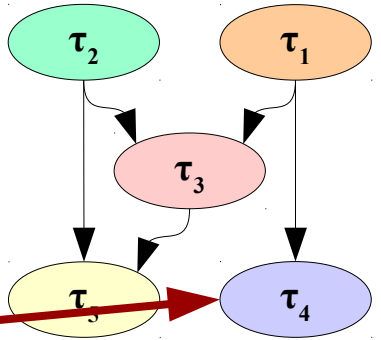
ALGORITHM

for all: **i = n … 1** // iterate over all tasks, from lowest priority
  $SA_i = 0$ // set initial stack address to 0
  for all: **j > i** // iterate over all tasks with lower priority
    if τ_i **can preempt** τ_j *then*
      $SA_i = \max(SA_i, SA_j + SU_j)$

Computing the Stack Addresses

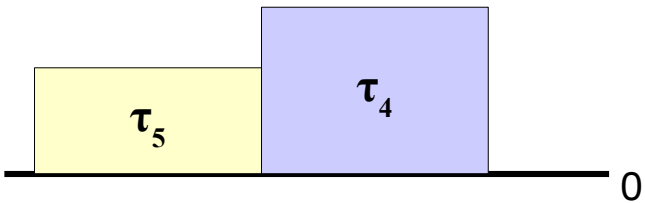
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

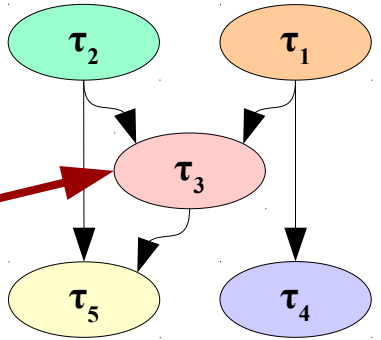
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
$\mathbf{SA_i = 0}$ // set initial stack address to 0
for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
if τ_i **can preempt** τ_j *then*
$\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

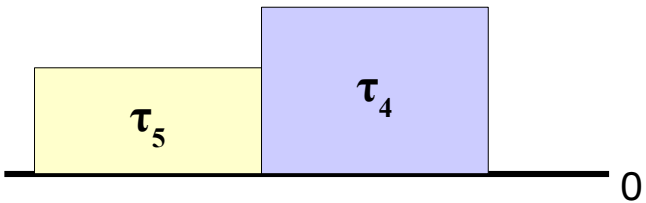
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

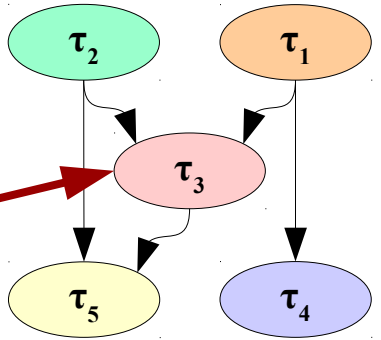
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
$\mathbf{SA_i = 0}$ // set initial stack address to 0
for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
if τ_i **can preempt** τ_j *then*
$\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

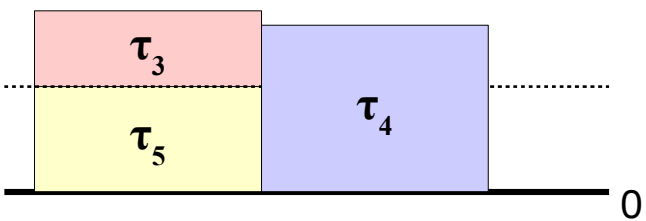
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

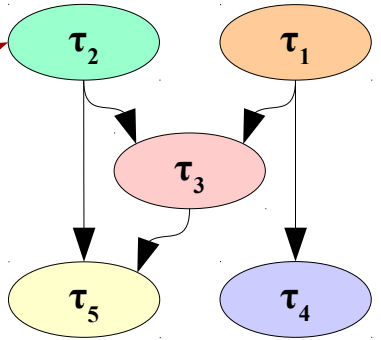
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
  $\mathbf{SA_i = 0}$ // set initial stack address to 0
  for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
    if τ_i **can preempt** τ_j *then*
      $\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

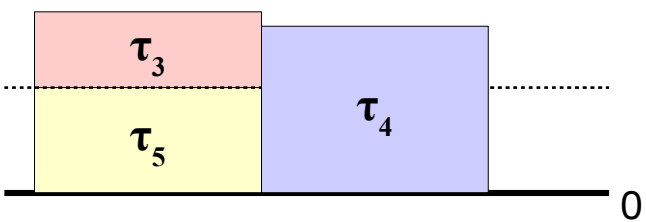
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

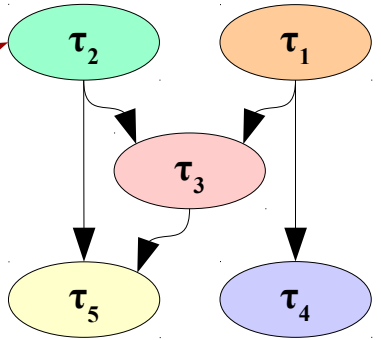
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
$\mathbf{SA_i = 0}$ // set initial stack address to 0
for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
if τ_i **can preempt** τ_j *then*
$\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

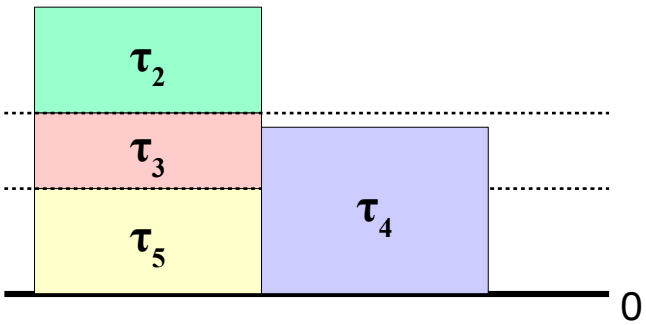
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

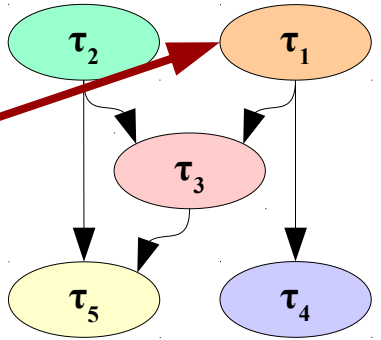
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
  $\mathbf{SA_i = 0}$ // set initial stack address to 0
  for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
    if τ_i **can preempt** τ_j *then*
      $\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

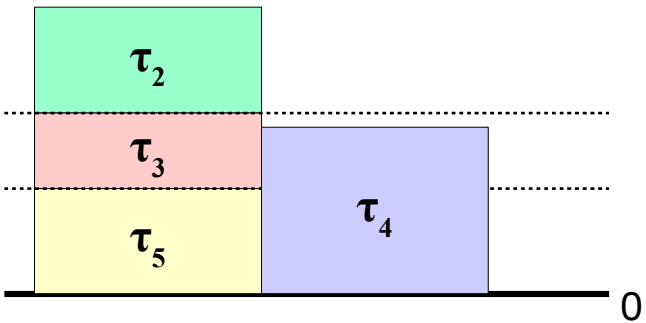
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

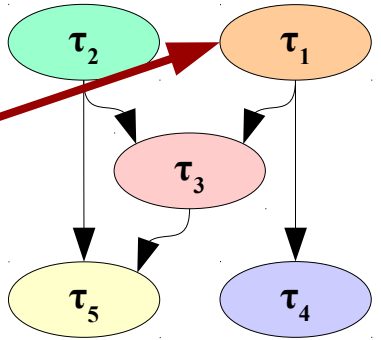
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
  $\mathbf{SA_i = 0}$ // set initial stack address to 0
  for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
    if τ_i **can preempt** τ_j *then*
      $\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

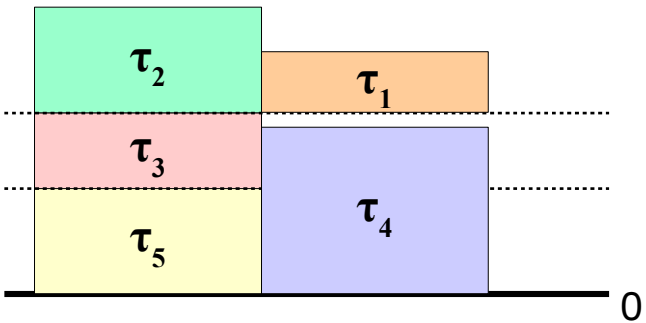
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



ALGORITHM

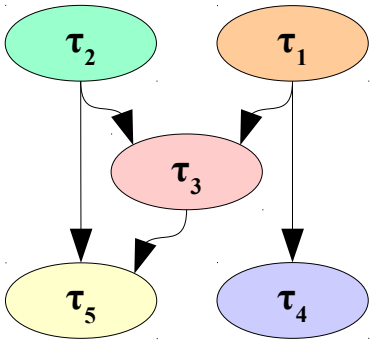
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
  $\mathbf{SA_i = 0}$ // set initial stack address to 0
  for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
    if τ_i **can preempt** τ_j *then*
      $\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$



Computing the Stack Addresses

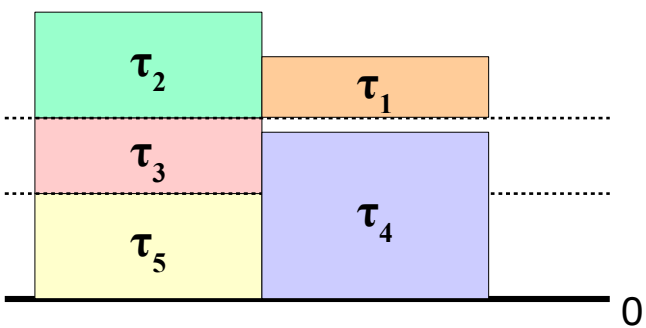
Select worst-case stack pointer **SA** based on

- preemption graph
- SU = max Stack usage of a task



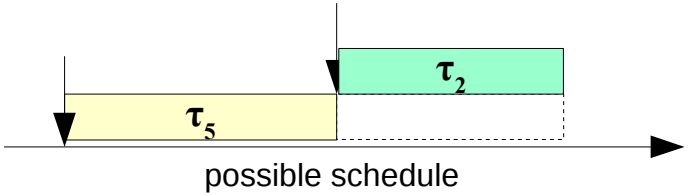
ALGORITHM

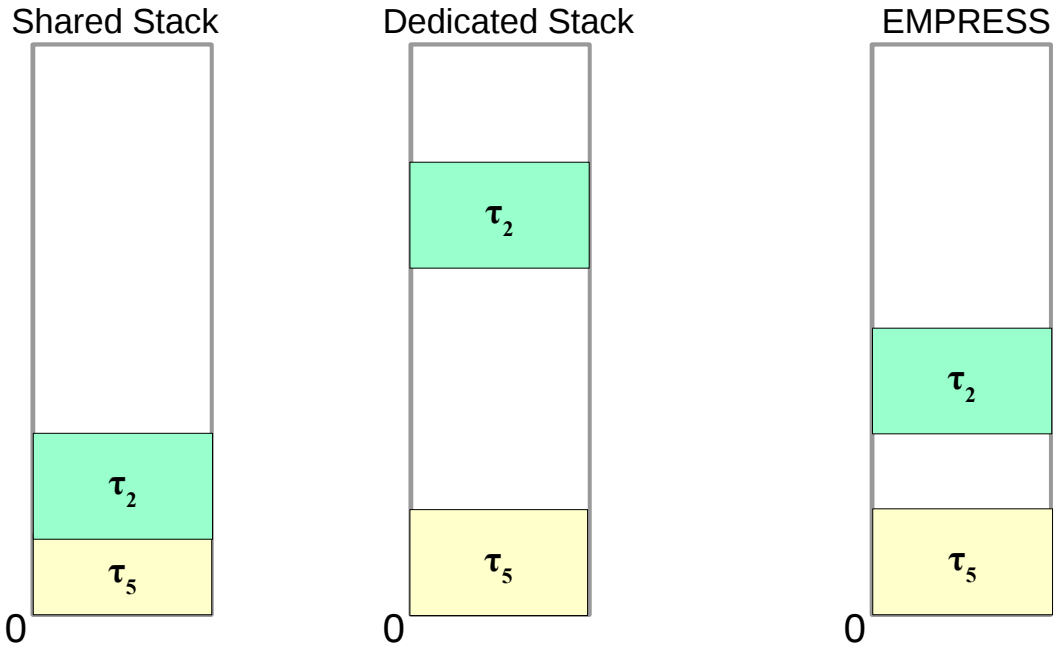
for all: $\mathbf{i = n \ldots 1}$ // iterate over all tasks, from lowest priority
$\mathbf{SA_i = 0}$ // set initial stack address to 0
for all: $\mathbf{j > i}$ // iterate over all tasks with lower priority
if τ_i **can preempt** τ_j *then*
$\mathbf{SA_i = max\ (SA_i, SA_j + SU_j)}$

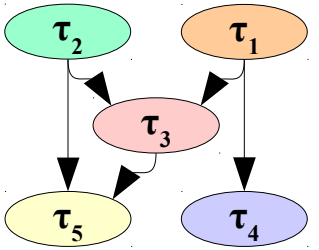


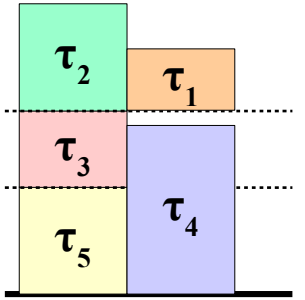
Similar algorithm for computing worst-case stack usage for shared stacks [8]

Difference to other Implementations (1)

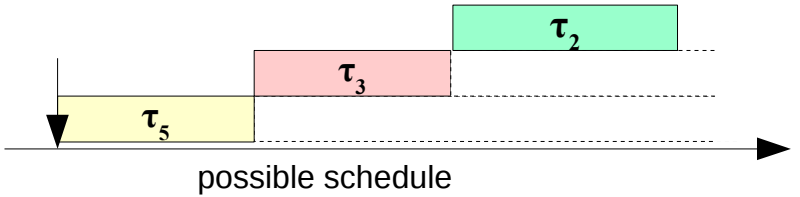


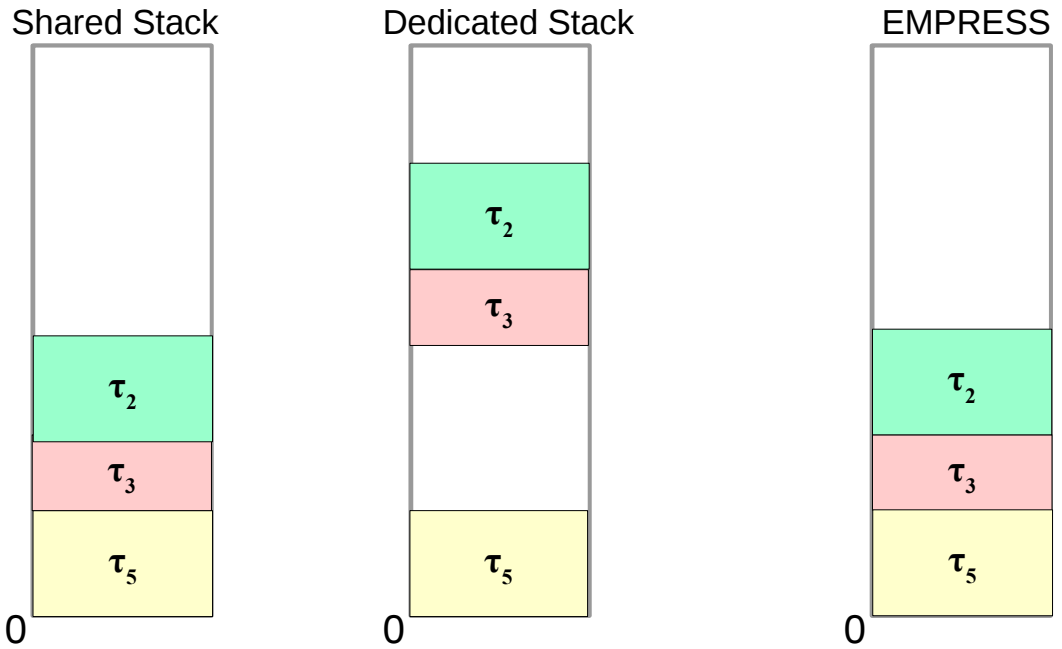


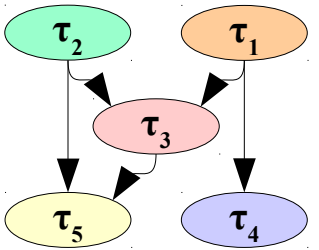


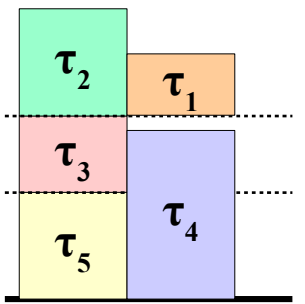


Difference to other Implementations (2)

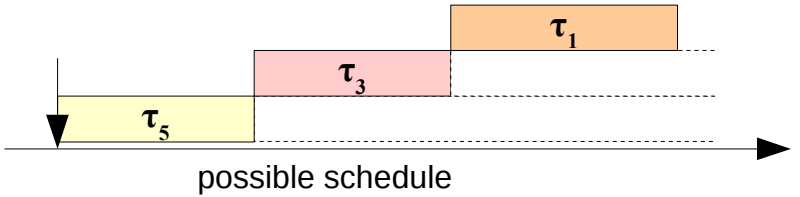


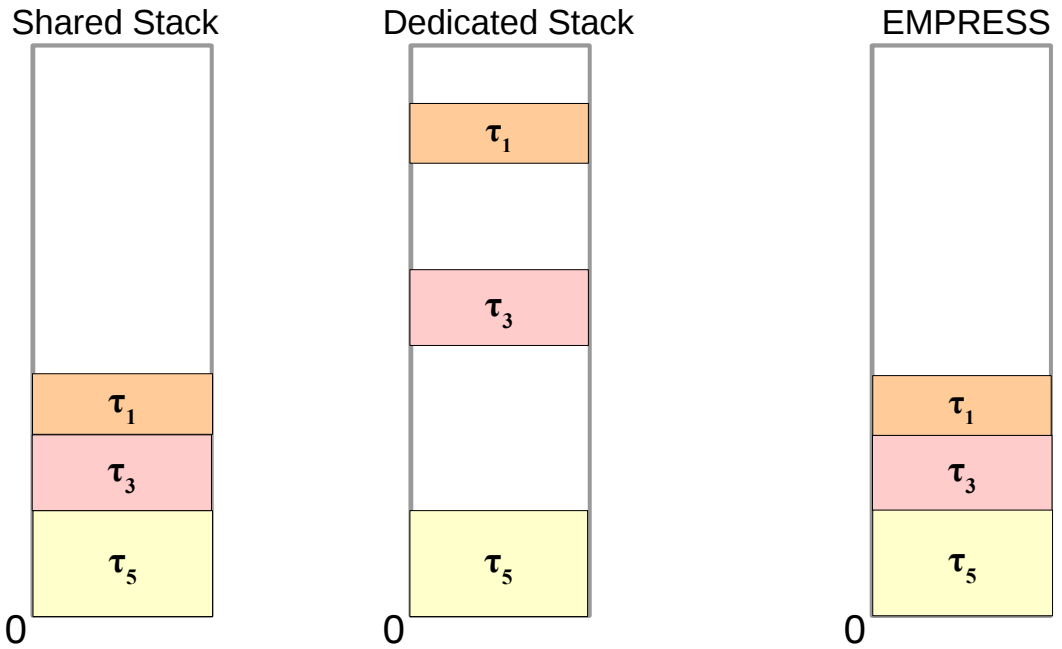


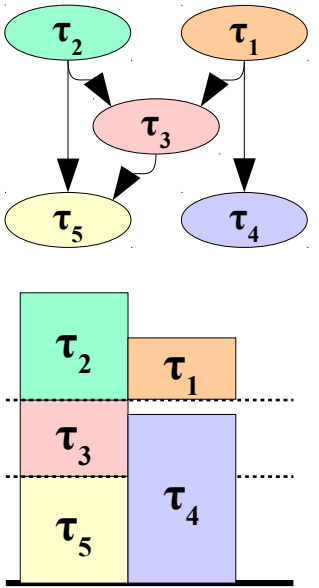




Difference to other Implementations (3)







Content

1) System Assumptions
2) Current Stack Implementations
3) EMPRESS
4) **Evaluation/Case Study**
5) Conclusions

Case Study: PapaBench

Control software of an UAV [28]

- C-Code available
- Complete task-set definition (deadlines, periods, precedence)
- 2 task-sets *Fly-by-wire* (5 tasks) / *Autopilot* (8 tasks)

Evaluation of

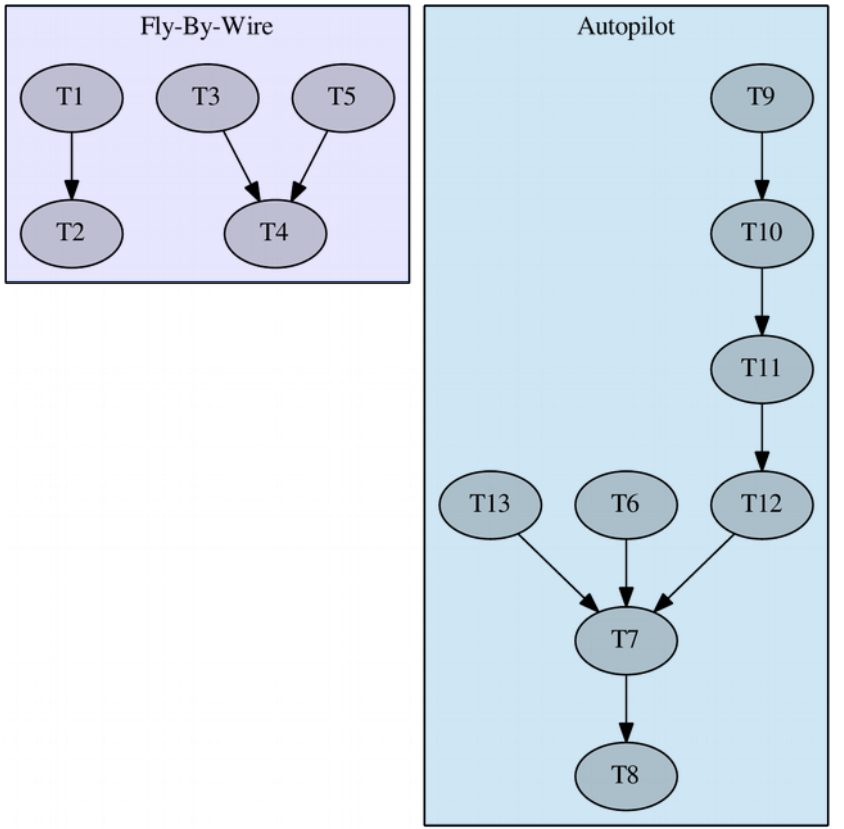
1) Reduction of Stack Usage
2) Impact on Predictability

Case Study: PapaBench

	Task	Description	Period	Priority	Stack Usage (Byte)*
Fly-By-Wire	T1	Receive Radio-Command	25ms	1	48
	T2	Send Data to MCU0	25ms	2	24
	T3	Receive MCU0 values	50ms	3	48
	T4	Transmit Servos	50ms	4	16
	T5	Check Failsafe	50ms	5	48
	T6	Managing Radio orders	25ms	1	120
	T7	Stabilization	50ms	2	72
Autopilot	T8	Send Data to MCU1	50ms	3	0
	T9	Receive GPS Data	250ms	5	128
	T10	Navigation	250ms	6	188
	T11	Altitude Control	250ms	7	56
	T12	Climb Control	250ms	8	72
	T13	Reporting Task	100ms	4	44

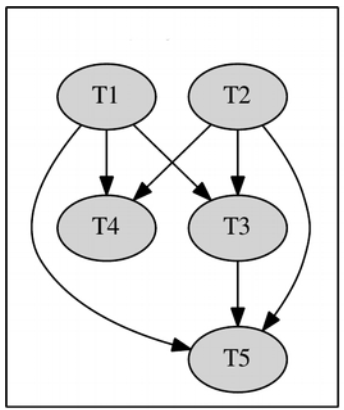
* derived by Absint's Static Stack Analyzer [17,22]

Precedence Constraints

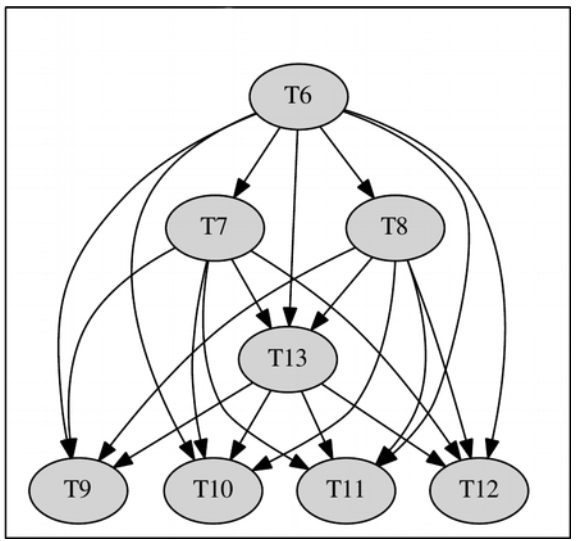


Case Study: PapaBench
Reduction of Stack Usage

Preemption Graph



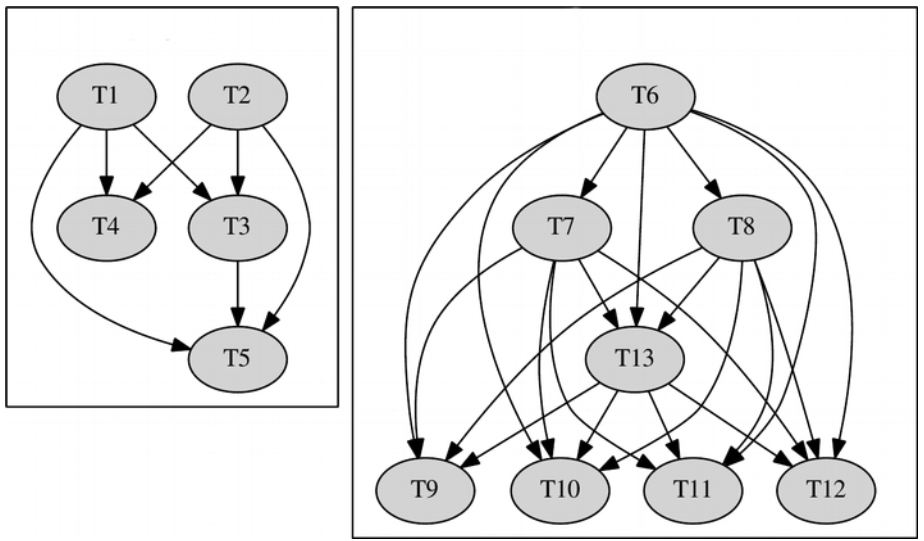
Fly-By-Wire



Autopilot

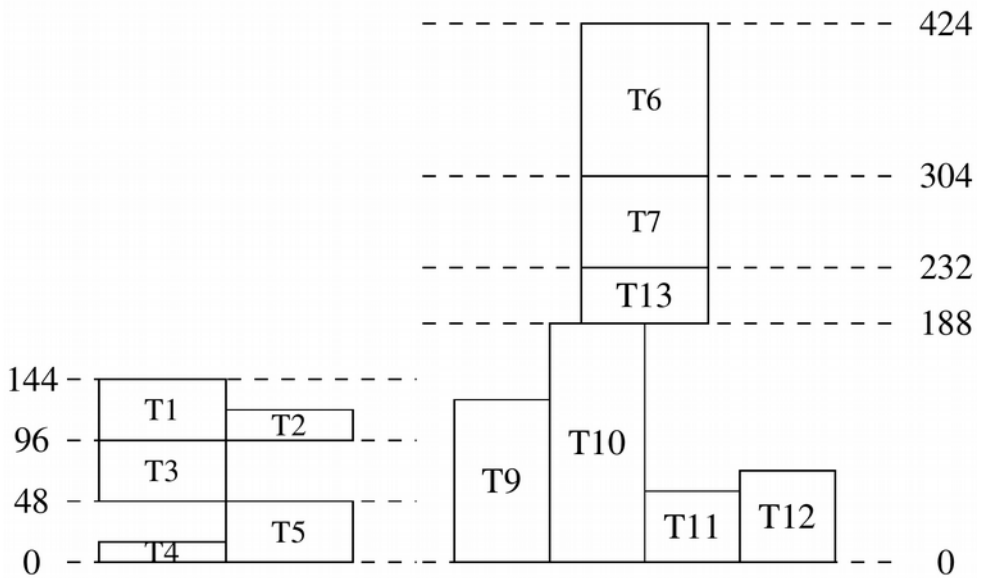
Case Study: PapaBench
Reduction of Stack Usage

Preemption Graph



Fly-By-Wire Autopilot

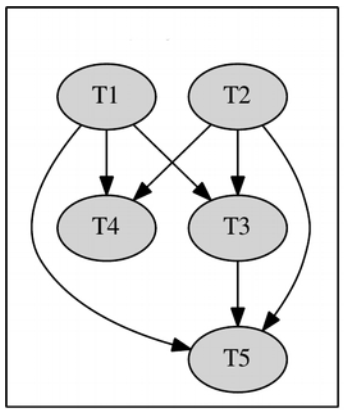
EMPRESS Stack Layout



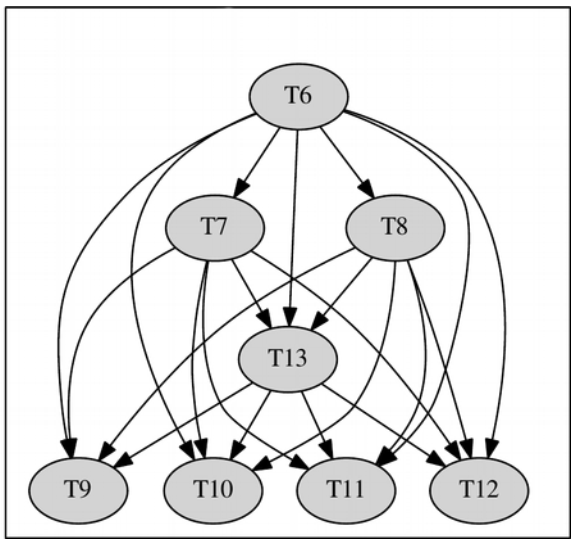
Fly-By-Wire Autopilot

Case Study: PapaBench
Reduction of Stack Usage

Preemption Graph

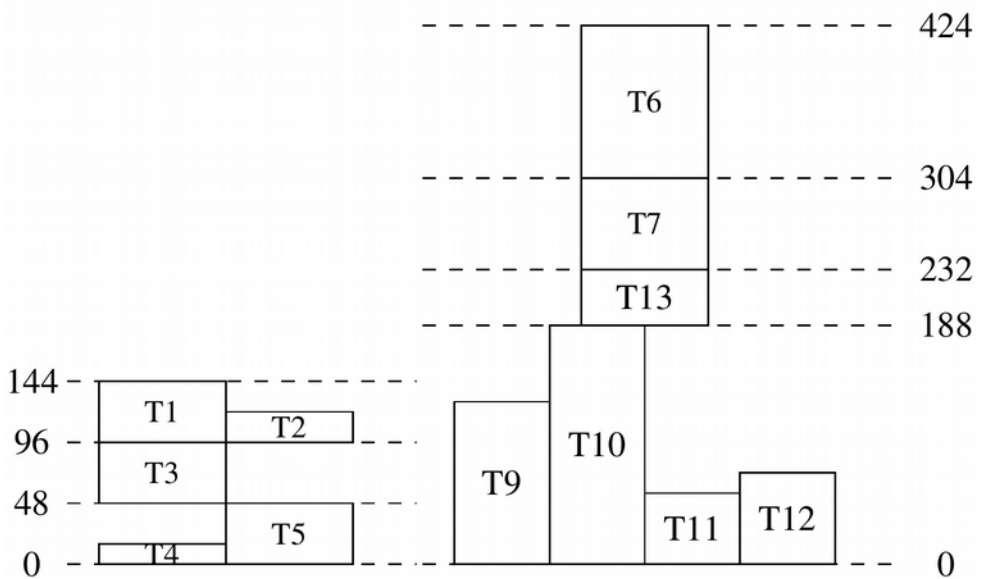


Fly-By-Wire



Autopilot

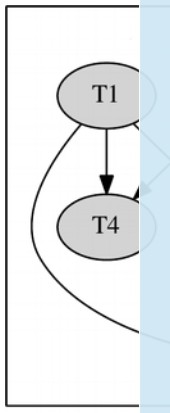
EMPRESS Stack Layout



Fly-By-Wire Autopilot

	Dedicated Stacks	Shared Stack	EMPRESS
Fly-By-Wire	184	**144 (-21%)**	
Autopilot	680	**424 (-37%)**	

Case Study: PapaBench
Reduction of Stack Usage



General Case (Dedicated Stack vs. Shared Stack/EMPRESS)

- No reduction if fully preemptive/no precedence constraints
  … unrealistic
- Real systems plenty of precedence constraints/resource sharing (ECRTS 2017 Industrial Challenge [25])
  - 21% and 37% probably on the low side
- Precedence constraints/FPTS/Non-preemptive regions to reduce stack size up to 75% [10]

Case Study: PapaBench
Improved Predictability

Timing Analysis + Shared Stack/Variable Stack pointer

What could we do to analyze with unknown Stack Pointer?

1) Perform n analyses:
   - one for each potential stack pointer
   - not scalable; for PapaBench: 300 vs 13 analyses

2) Perform one imprecise analysis

Case Study: PapaBench
Improved Predictability

Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

Case Study: PapaBench

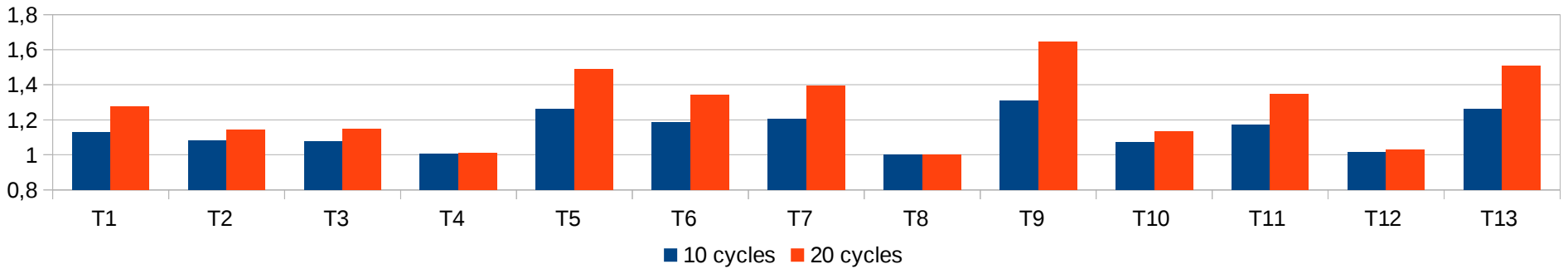
Improved Predictability

Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)



Relative increase in WCET bound due to variable stack pointer

Case Study: PapaBench
Improved Predictability

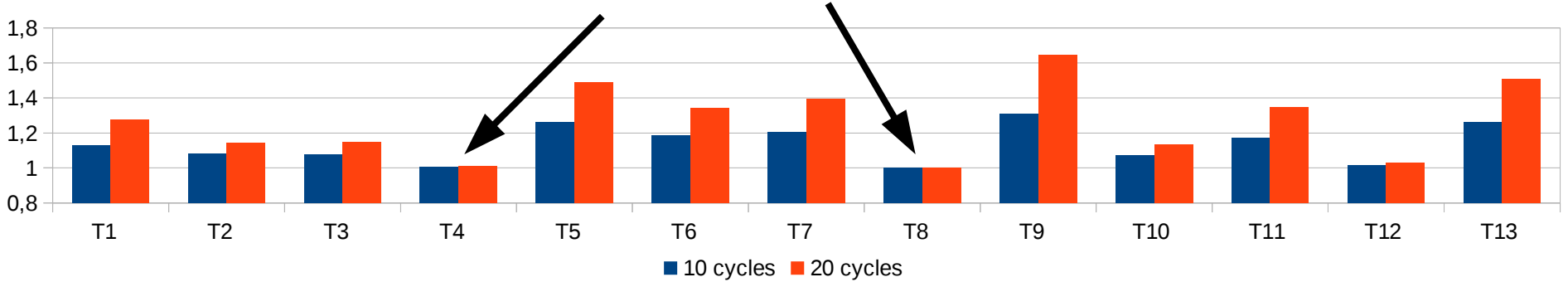
Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

small stack usage/no reuse of stack data



Relative increase in WCET bound due to variable stack pointer

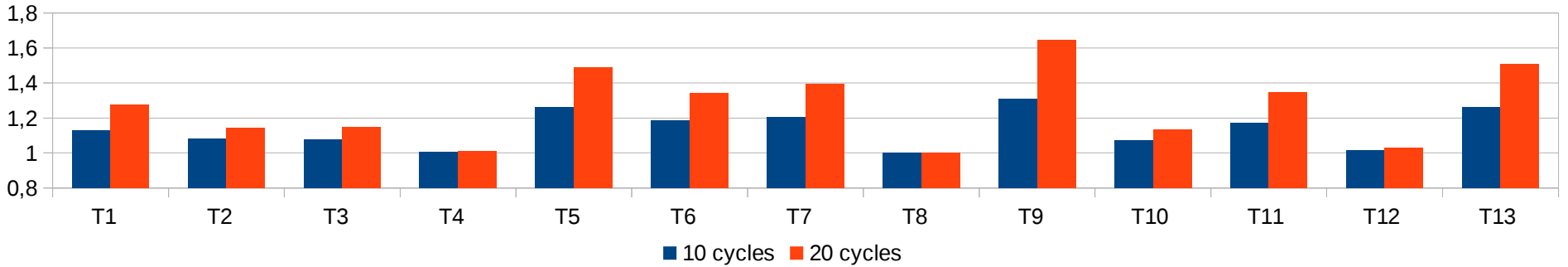
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)



Relative increase in WCET bound due to variable stack pointer

Case Study: PapaBench

Improved Predictability

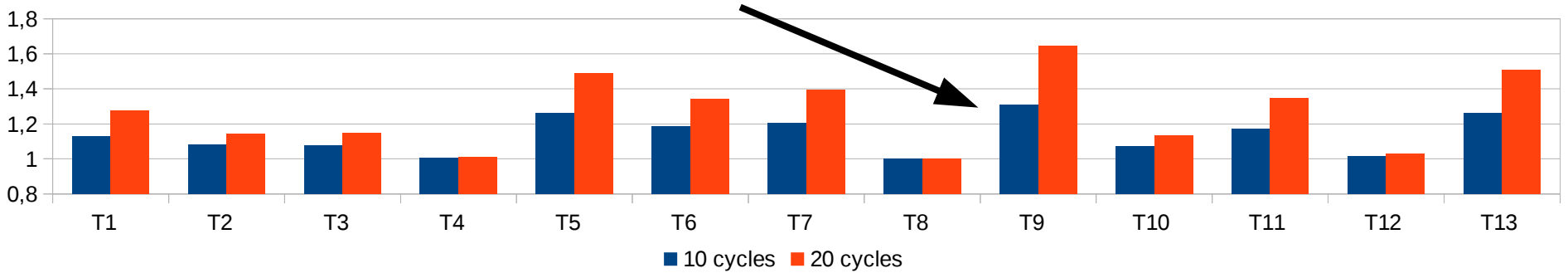
Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

highest increase: 30% (10 cyc), 60% (20 cyc), 2nd highest stack usage



Relative increase in WCET bound due to variable stack pointer

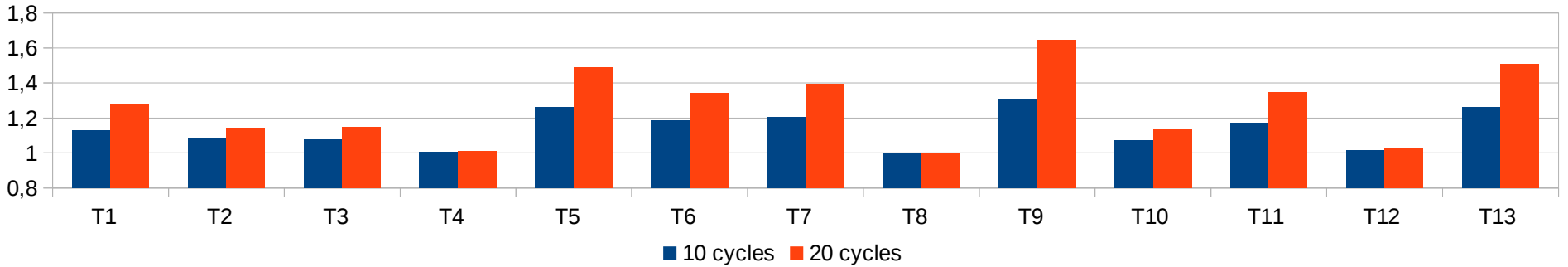
Case Study: PapaBench
Improved Predictability

Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)



Relative increase in WCET bound due to variable stack pointer

Case Study: PapaBench

Improved Predictability

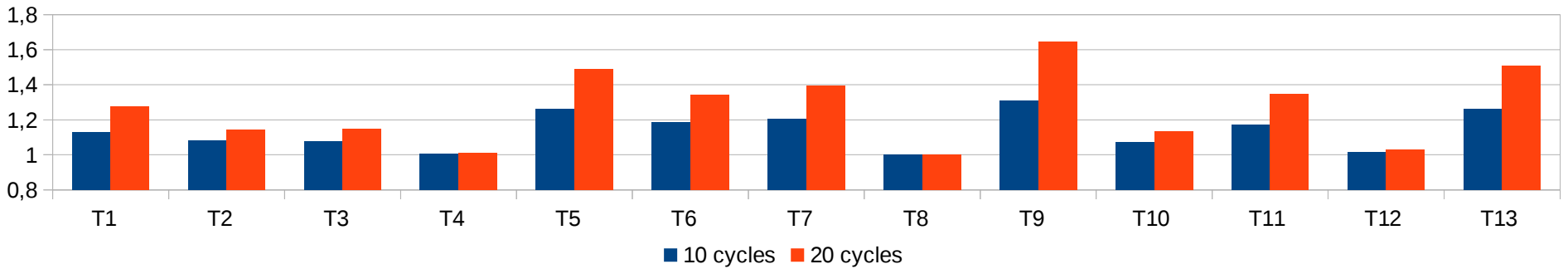
Architecture: ARMv7

- Instruction Scratchpad
- Data Cache (Size: 2kB, 4way LRU, 16Byte Linesize, 32 sets)
- Memory access time: 10 cycles/20 cycles

Timing Analysis via **Absint's Timing Profiler**

1) Static Stack Pointer (EMPRESS/Dedicated Stack, WCET normalized to 1)
2) Range of Stack Pointer (Shared Stacks)

average increase: 6% (10 cyc), 18% (20 cyc)



Relative increase in WCET bound due to variable stack pointer

Implementation of EMPRESS

Is EMPRESS interesting for RTOS vendors/industry?

Can we implement EMPRESS within an RTOS?

What is the implementation overhead?

Implementation of EMPRESS

Is EMPRESS interesting for RTOS vendors/industry?

Can we implement EMPRESS within an RTOS?

What is the implementation overhead?

Implementation of EMPRESS

Is EMPRESS interesting for RTOS vendors/industry? **Yes.**

Can we implement EMPRESS within an RTOS? **Sure.**

What is the implementation overhead? **Depends.**

EMPRESS within Erika RTOS [13]

Implementation requirements:

- Absence of blocking primitives
- Mutexes handled via Immediate Priority Ceiling Protocols [29]
- Run-to-completion semantics

ERIKA Enterprise v2

- Additional stack saving/stack pointer modification

Overhead:

- 2 instructions at task activation

ERIKA Enterprise v3

- Already offers possibility to overlay stack regions with shared stacks

Overhead:

- No additional costs/zero overhead

Content

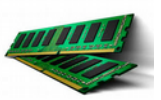
1) System Assumptions
2) Current Stack Implementations
3) EMPRESS
4) Evaluation/Case Study
5) **Conclusions**

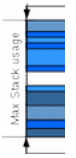
Conclusions

EMPRESS: an Efficient and effective Method for PREdictable Stack Sharing

Reconciling **Predictability** and **Performance**:

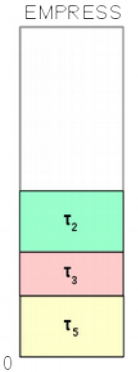
- enables precise **timing verification**
- enables **optimization of memory/cache layout**
- reduces **stack usage**
- limits **runtime overheads**





Idea:

- Stack sharing of mutually non-preemptive tasks
- Static Stack Pointer = Worst-case Stack Pointer of shared stack



Bibliography

[8] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. **Stack size analysis for interrupt-driven programs.** In Proc. 10th International Symposium on Static Analysis (SAS), pages 109–126, June 2003.

[10] R.I. Davis, N. Merriam, and N.J. Tracey. **How embedded applications using an RTOS can stay within on-chip memory limits.** In Proc. Work in Progress and Industrial Experience Session, 12th Euromicro Conference on Real- Time Systems (ECRTS), pages 43–50, June 2000.

[12] C. Ferdinand and R. Heckmann. **aiT: worst case execution time prediction by static program analysis.** In Proc. International Federation for Information Processing (IFIP), volume 156, pages 377–384, Aug. 2004.

[13] P. Gai, E. Bini, G. Lipari, M. Di Natale, and L. Abeni. **Architecture for a portable open source real time kernel environment.** In Proc. 2nd Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial, Nov. 2000.

[17] **AbsInt Angewandte Informatik GmbH. Static Stack Analyzer.** https://www.absint.com/stackanalyzer/index.htm, 2018.

[22] D. Kästner and C. Ferdinand. **Proving the absence of stack overflows.** In Proc. 33rd International Conference on Computer Safety, Reliability, and Security (SAFE-COMP), pages 202–213, Sep. 2014.

[25] S. Kramer, D. Ziegenbein, and A. Hamann. **Real world automotive benchmark for free.** In Proc. 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), July 2015.

[28] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. **PapaBench: a Free Real-Time Benchmark.** In Proc. 6th International Workshop on Worst-Case Execution Time Analysis (WCET), July 2006.

[29] OSEK group. **OSEK/VDX operating system. Technical report**, February 2005. OSEK OS 2.2.3 available as ISO Standard 17356-3 2005.