Evolving Scheduling Strategies for Multi-Processor Real-Time Systems

Frank Feinbube, *Max Plauth*, Christian Kieschnick and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute, Germany

Motivation & Background

- Real-Time scheduling on multi-processor system is a much harder problem than RT scheduling on uni-processor systems

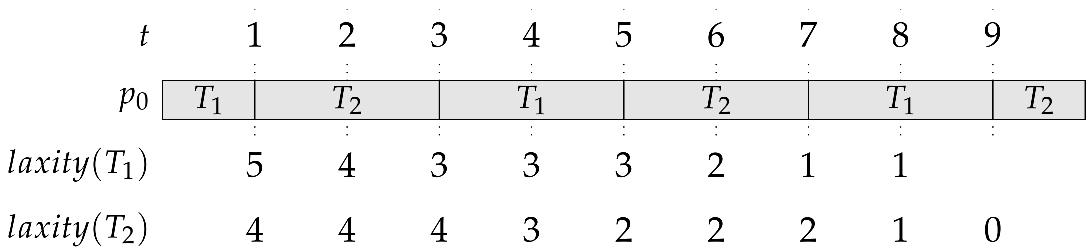
- Uni-processor systems:
  - Earliest Deadline First has been proven to be the best algorithm to guarantee the correct execution of prioritized tasks

- Multi-processor systems:
  - Uni-processor scheduling approaches are not feasible for multi-processor systems anymore

Motivation & Background

- No optimal, priority-driven algorithm exists for arbitrary task sets
  - Optimal algorithms only exists for periodic task sets (e.g. laxity driven)
  - Most algorithms ignore task migrations in their cost-model
    - performance often remains insufficient in practice
  - No optimal algorithm exists for the general case (Fisher 2007)

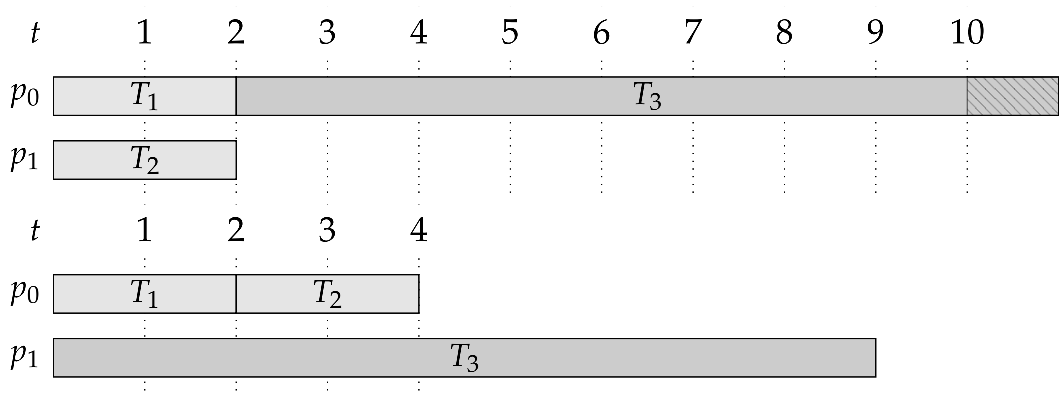
	A	AD	C
T_1	0	10	5
T_2	0	10	5



Motivation & Background

- Dhall's effect
  - Although it is possible to schedule all tasks according to their deadline, Earliest Deadline First fails to do so

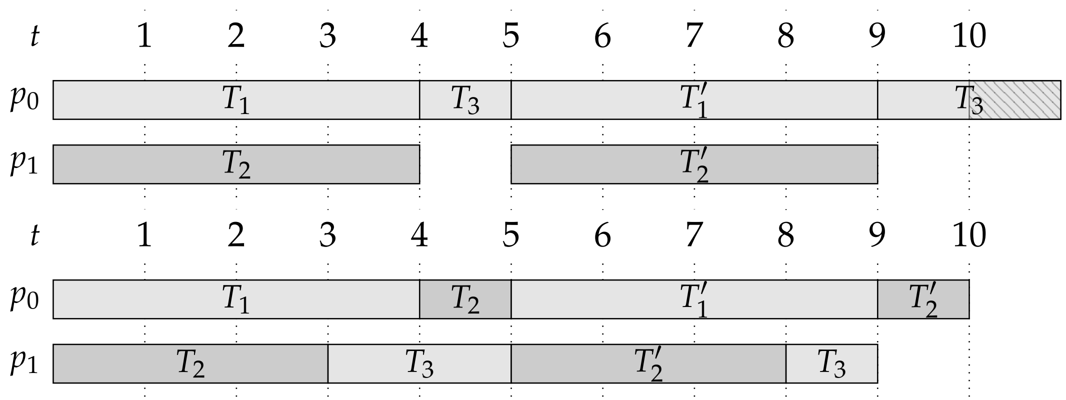
	A	AD	C
T_1	0	9	2
T_2	0	9	2
T_3	0	10	9



Motivation & Background

- Levin's pure global task sets:
  - Although it is possible to schedule all tasks according to their deadline, it is impossible to do so by pinning tasks to a single processor

	A	AD	C
T_1	0	5	4
T_2	0	5	4
T_3	0	10	3

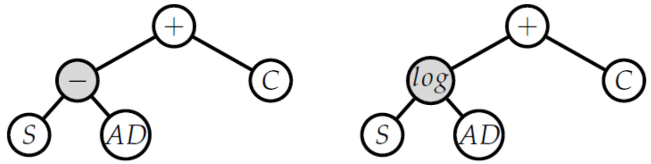


Research Gap

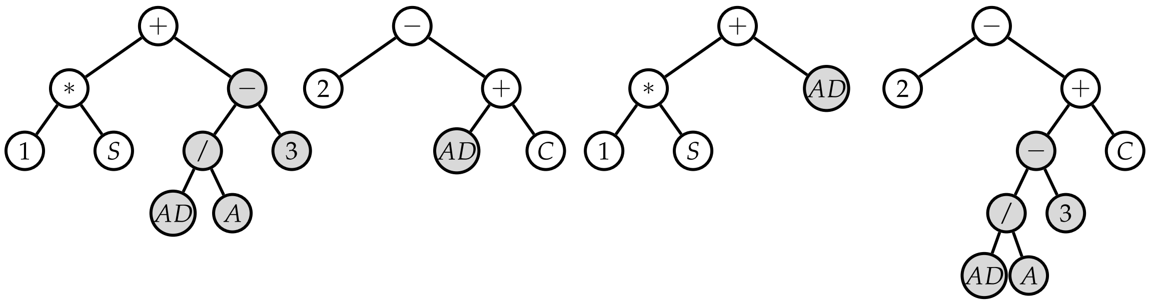
- Identify novel algorithms by exploring the solution space for real-time scheduling algorithms.
- Create algorithms complying with desired characteristics such as the number of task migrations and maximal system utilization.

Approach

- Application of genetic programming to prioritization functions
- Functions are represented as trees of operands and terminals
- Mutation: random nodes are replaced



- Breeding: sub-trees get swapped



Approach

- Evaluation of fitness
  - Executability
  - Number of migrations
  - Multi-goal optimization

- Selection process
  - Tournament mode (2, 4, 6 or 8 participants)
    - Larger selection pressure yields executable strategies quicker

- Overfitting
  - Usually considered as a weakness
  - Is able to create optimal scheduling strategies for specific workloads

Approach

- Evaluation of fitness requires test task sets
- Strategy 1: attempt generation of “complete” task sets
  - Feasible only for small number of CPUs and quanten
  - 8 processors, 6 quanta intervals → 10^8 task sets

- Strategy 2: compile representative task sets from literature
  - It is hard to find real, global task sets

- Problem size classes
- Q_1: 1, 2, 4 processors
- Q_{10}: 10, 20, 40 processors
- Q_{100}: 100, 200, 400 processors

Approach

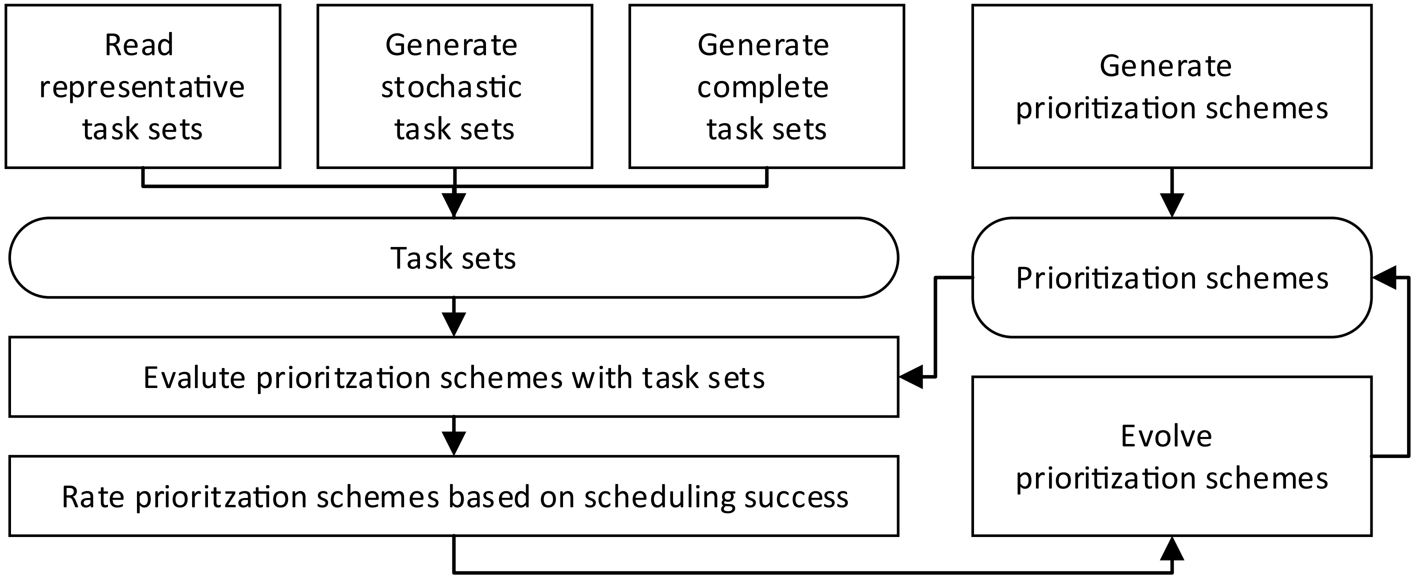
- Main training set Q_A
  - Dhall (5 variants)
  - RMS3 (3 variants)
  - Lemma3 (9 variants)
  - Partitioned (5 variants)
  - WikiRMS (3 variants)
- Counter balancing training set Q_B
  - Dhall (2 variants)
  - SlackDhall (3 variants)
  - RMS3 (2 variants)
  - RMS4 (2 variants)
  - Detail (1 variants)
  - Lemma3 (3 variants)
  - Partioned (2 variants)
  - WikiRMS (2 variants)
  - Interwoven (2 variants)
  - Levin (1 variants)

Approach

- Main training set Q_A
  - Dhall (5 variants)
  - RMS3 (3 variants)
  - Lemma3 (9 variants)
  - Partitioned (5 variants)
  - WikiRMS (3 variants)
- Counter balancing training set Q_B
  - Dhall (2 variants), SlackDhall (3 variants)
  - RMS3 (2 variants), RMS4 (2 variants)
  - Detail (1 variants), Lemma3 (3 variants)
  - Partioned (2 variants), WikiRMS (2 variants)
  - Interwoven (2 variants), Levin (1 variants)

	periodic	partitionable	Laxity-based	global EDF	EDF-US	EDZL
RMS3	✓	✓		2*		
RMS4	✓	✓		2* 4 8 16		4* 8* 16*
WikiEDF	✓	✓				
Partitioned	✓	✓	2* 4* 8* 16*	4* 8* 16*	2* 4* 8* 16*	4* 8* 16*
Dhall		✓		2 4 8 16	1*	
SlackDhall		✓	4* 8* 16*		1* 2* 4* 8* 16*	4* 8* 16*
Detail		✓		2		
Split		✓				
Interwoven		✓	2 4 8 16	2 4 8 16	1 2 4 8 16	2 4 8 16
Levin [11]	✓		2 4 8 16	2 4 8 16	2 4 8 16	2 4 8 16

Approach



Read representative task sets
Generate stochastic task sets
Generate complete task sets
Generate prioritization schemes
Task sets
Prioritization schemes
Evalute prioritzation schemes with task sets
Evolve prioritization schemes
Rate prioritization schemes based on scheduling success

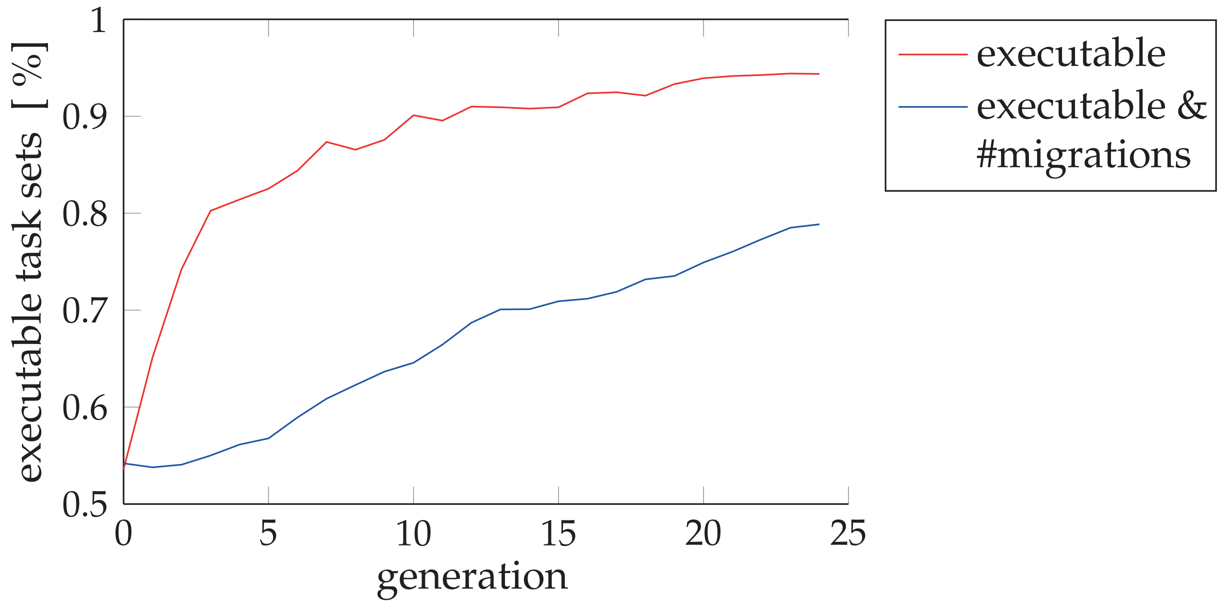
Approach

- Generic scheduler for the simulation framework

```
for(runtime = 0;
        runtime < simulationEnd && !missedDeadline(tasks);
        ++runtime)
{
        activeTasks = filterActive(tasks);

        // this is exchanged with each prioritization scheme
        prioritizationScheme->prioritizeTasks(activeTasks);

        orderDescendantByPriority(activeTasks);
        tasksToSchedule = selectFirst(activeTasks, processors);

        simulateDiscreteStep(tasksToSchedule);
}
```

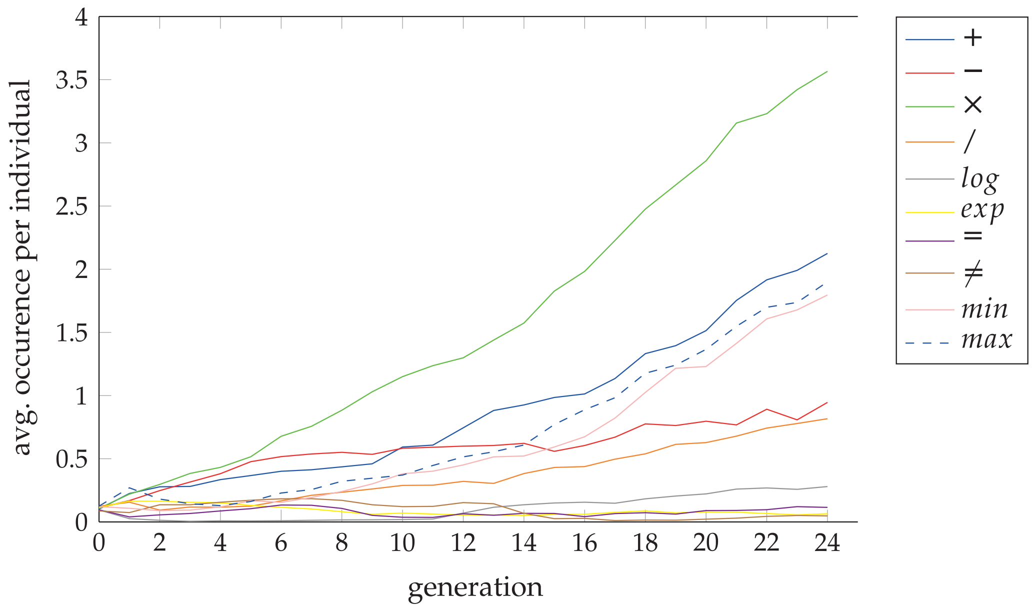
Qualitative Evaluation

- Fitness ratings that are based on the number of executable task sets exclusively show a faster evolutionary progress, but introduce a considerable amount of task migrations.



Qualitative Evaluation

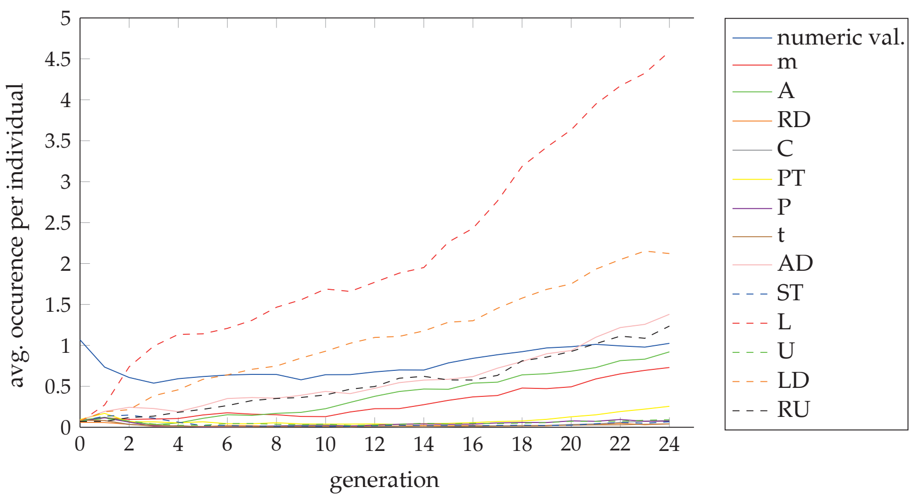
- The evolutionary approach favored fundamental arithmetic operations and the min/max functions.
- Complex operations such as log, exp and equals were less succesful



Qualitative Evaluation

- Terminals with dynamic properties such as Laxity L, remaining execution time LD and remaining utilization RU were especially successful in the evolutionary process

x	random floating point values from -10.0 to 10.0
0, 1	constant values 0 and 1
m	number of processors
A	arrival time
RD	relative deadline (relative to arrival time)
C	capacity = worst case execution time
PT	amount of C that has already been executed
P	current task priority (starting with 0)
T	current point in time
AD	absolute deadline = A + RD
ST	slack = RD - C
L	remaining surplus time = (AD - T) - (C - PT)
U	utilization created by task = C / RD
LD	remaining execution time = C - PT
RU	remaining utilization = LD / (AD - T)



Evolving Scheduling Strategies for Multi-Processor Real-Time Systems

Operating Systems & Middleware Group

Qualitative Evaluation

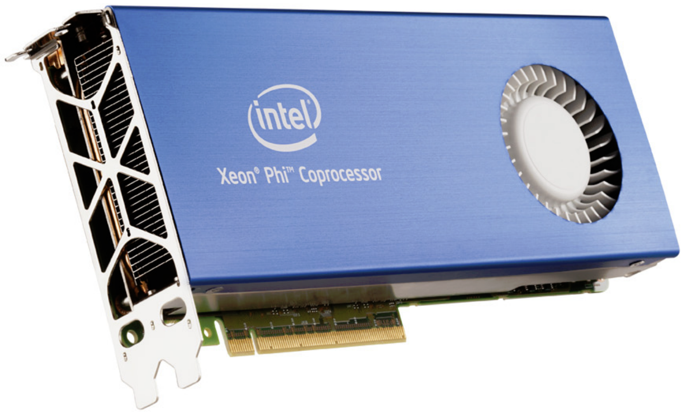
- Fittest prioritization functions by capability of scheduling task sets and the number of required task migrations:

function	# executable task sets		migrations / task set
L/RU	75	100 %	862
L	71	94.67 %	819
AD	56	74.67 %	24
$AD - 1.0$	56	74.67 %	24

Performance Evaluation

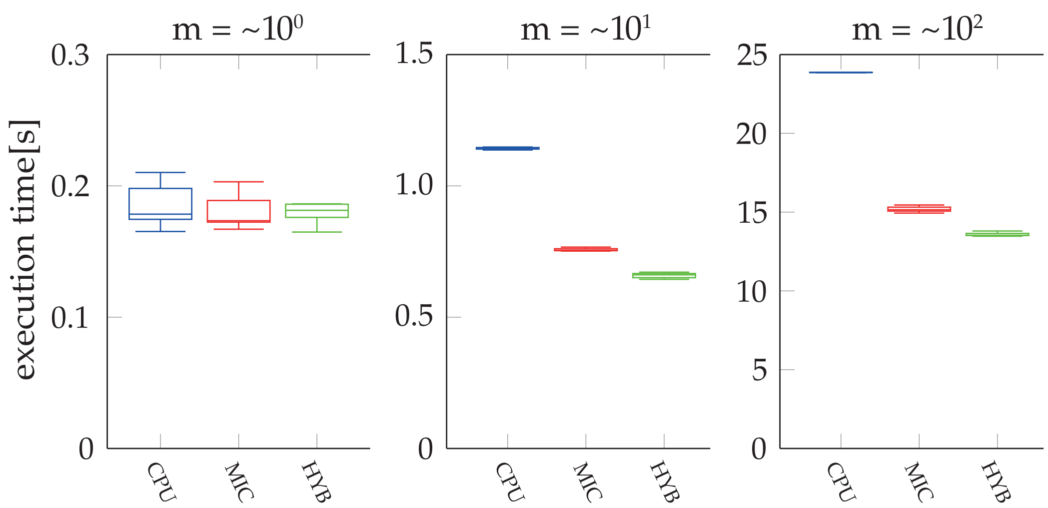
- Accelerator hardware: Xeon Phi 5110p
  - 60 Cores based on P54C architecture (Pentium)
  - 512 bit wide VPU per core
  - > 1.0 Ghz clock speed; 64bit based x86 instructions + SIMD
  - 1x 25 MB L2 Cache (=512KB per core) + 64 KB L1, Cache coherency
  - 8 GB of DDR5 on-board memory
  - 4 Hardware Threads per Core (240 logical cores)
    - Purpose: memory latency hiding
    - Switched after each instruction

- Host hardware: 2x Xeon E5620
  - 4 Cores each
  - 2.40 GHz
  - 25 GB main memory



Performance Evaluation

- Xeon Phi (MIC) always outperforms the CPU
  - Up to factor ~2x of speedup
  - Hybrid approach HYP always provides an additional performance
- Main bottleneck: few opportunities for vectorization



Conclusions

- For certain task sets, optimal prioritization functions were generated

- Overfitting can be leveraged to create optimal prioritization functions for well-known workloads
- Results harmonize well with Fisher's proof, that no priority-driven multicore scheduling algorithm exists for arbitrary tasksets

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Thank you
for your attention!

Frank Feinbube, *Max Plauth*, Christian Kieschnick and Andreas Polze
Operating Systems and Middleware Group
Hasso Plattner Institute, Germany