Supporting Global Resource Sharing in RUN-scheduled Multiprocessor Systems

L.Bonato, E.Mezzetti, T.Vardanega
University of Padua, Department of Mathematics

RTNS 2014, October 8-10

Our goal

Goal

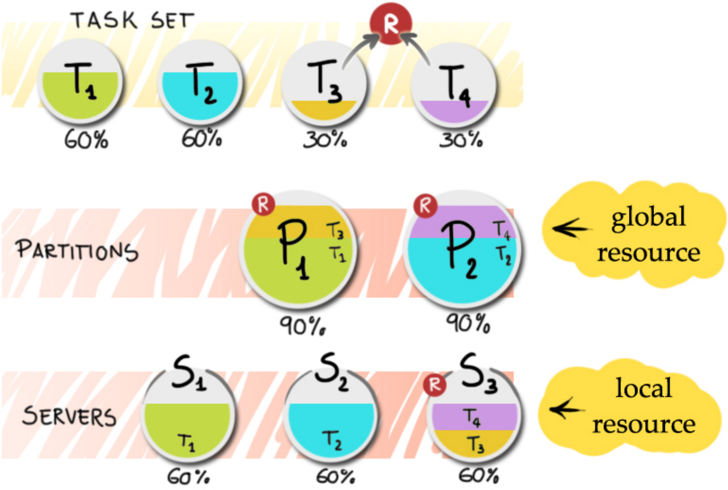
Extend RUN (Reduction to UNiprocessor) scheduling algorithm to not-independent tasks

Why RUN?

- optimal multiprocessor scheduling algorithm
- RUN uses servers, and servers can be convenient: logical packing vs feasibility packing

Are servers convenient when sharing resources?

assuming a platform with 2 processors...



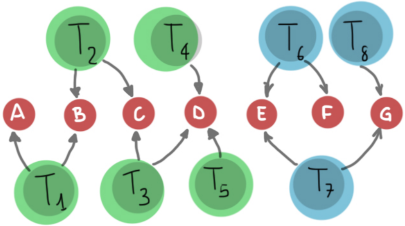
Our contribution

SBLP (Server Based Locking Protocol): a locking protocol for RUN.

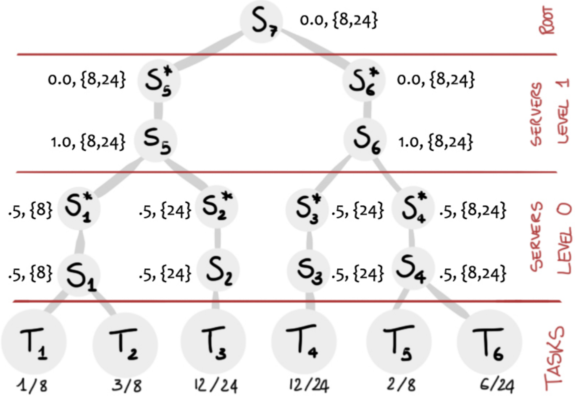
- "servers" as building block for isolating collaborative tasks: avoid bin-packing problem

collaborative tasks

We define two tasks to be collaborative if they belong to the same transitive closure formed on the relationship of sharing at least one common resource

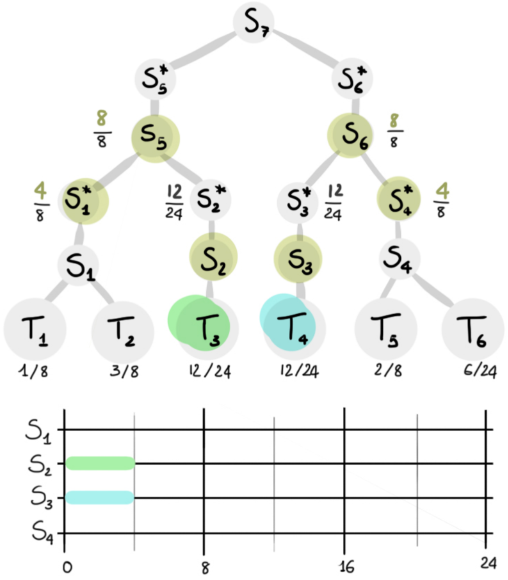


RUN: the reduction tree



- **tasks (T_j)** the starting point
- **packed server (S_i)** groups several tasks/servers together in a uniprocessor-like fashion
- **dual server (S_i^*)** represent the idle time of a packed server

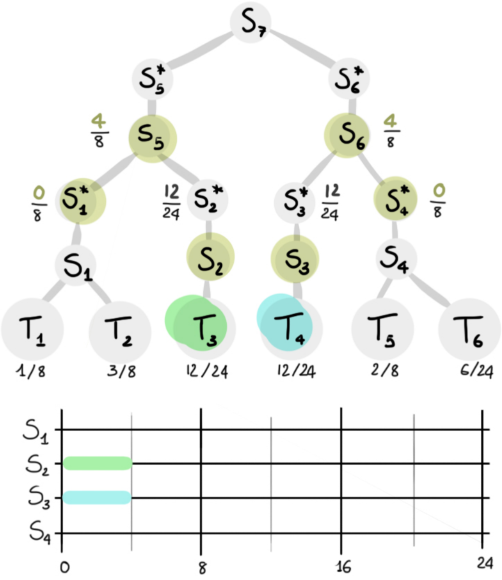
RUN executing



Scheduling decision propagates from the root to the leaves. E.g.:

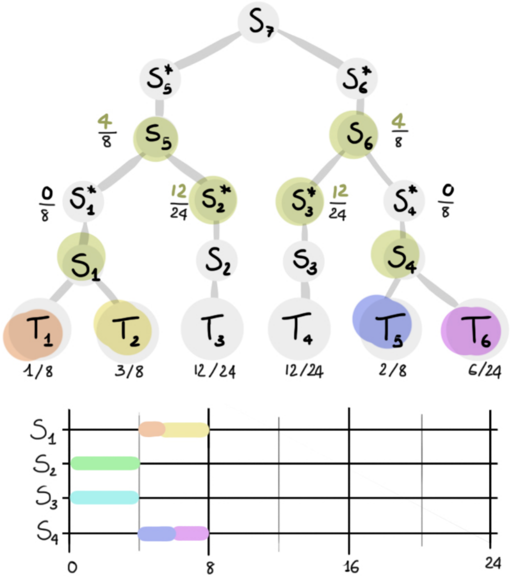
1. Root does not execute (no budget)
2. S_5^* does not execute because its parent does not
3. S_5 executes because its dual does not
4. S_1^* executes because client of S_5 with earliest deadline
5. S_1 does not execute because its dual does
6. T_1 and T_2 do not execute because S_1 does not

RUN executing



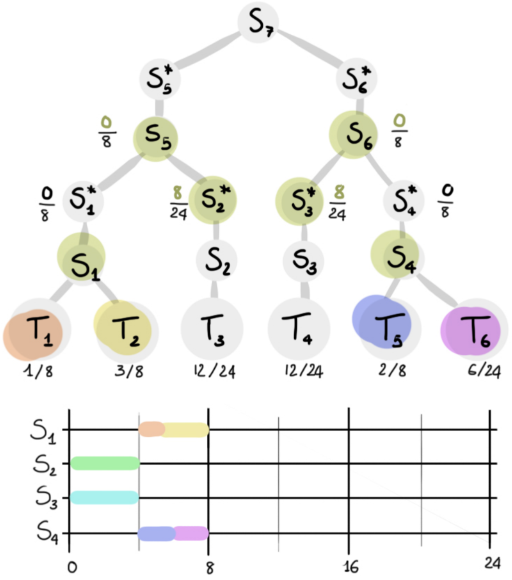
...and budgets are consumed during execution...

RUN executing



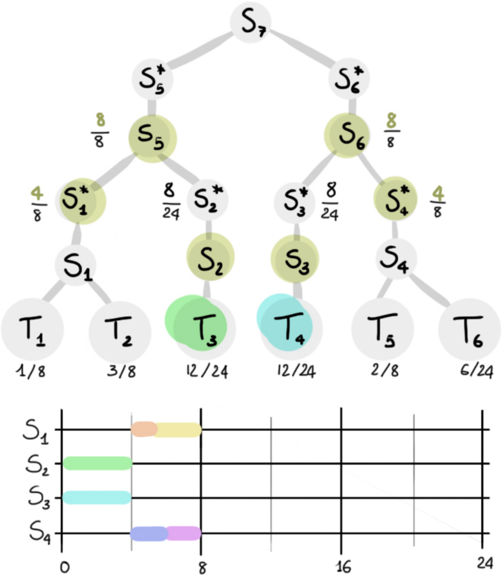
...and when budgets are exhausted, per-server EDF takes care to execute some other client...

RUN executing



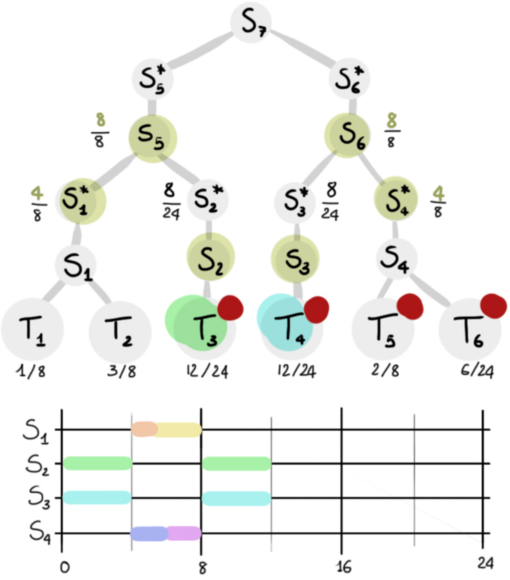
...and so on...

RUN executing



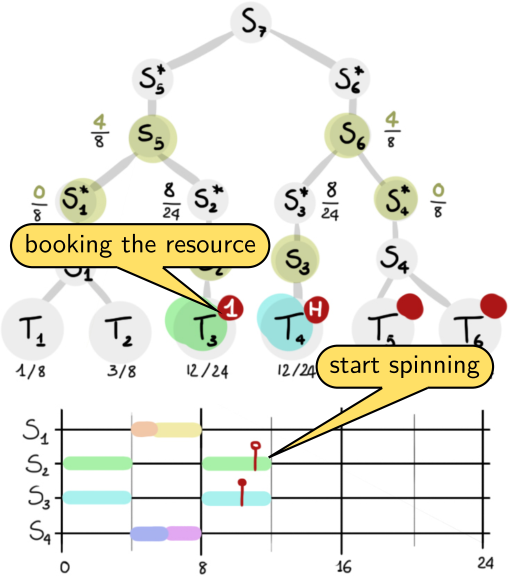
...until budgets are replenished when servers hit their deadlines

RUN+SBLP executing



…now assume tasks T_3, T_4, T_5 and T_6 share the same resource

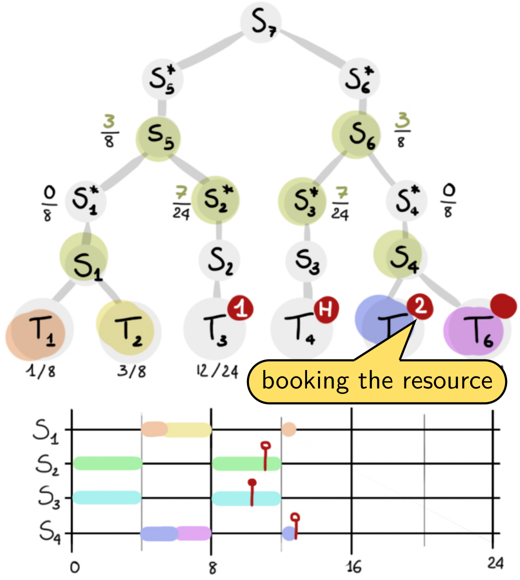
RUN+SBLP executing



some time before 12 both T_3 and T_4 request the resource

- T_4 locks the resource
- T_3 finds the resource locked and **books** the resource (appends in resource's FIFO queue)
- since the lock holder is executing, T_3 starts **spinning** while waiting for the resource to be released

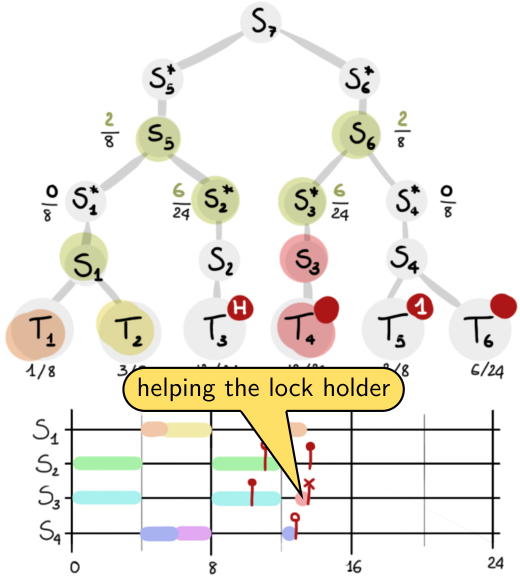
RUN+SBLP executing



a scheduling decision preempts T_3 and T_4.
When T_5 executes and try to lock the already locked resource:

- books the resource
- **lends** its processor to S_3 whose tasks is holding the resource

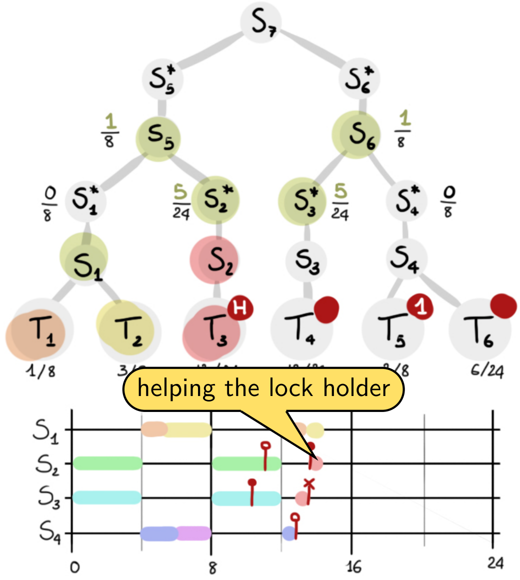
RUN+SBLP executing



T_5 does not spin because the lock holder *is not already executing*, but it lets execute T_4

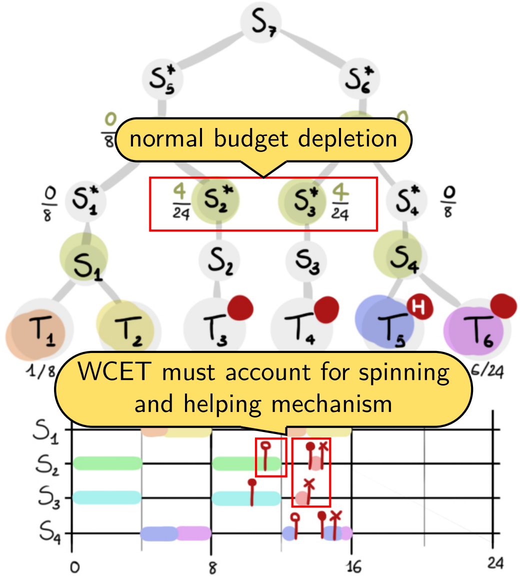
- resource released asap: no task wastes cpu-time by spinning if the holding task is not making any progress

RUN+SBLP executing



...and since requests must be served in FIFO order, T_3 becomes the new lock holder

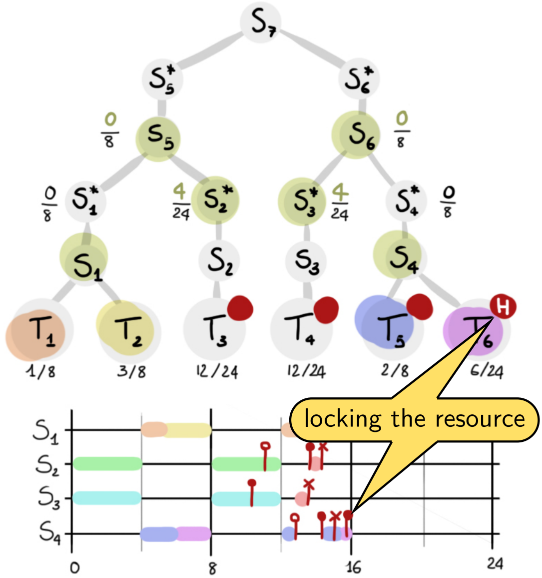
RUN+SBLP executing



budgets are consumed as if S_4 would never let other level-0 servers execute. **Increase WCET** of tasks to consider:

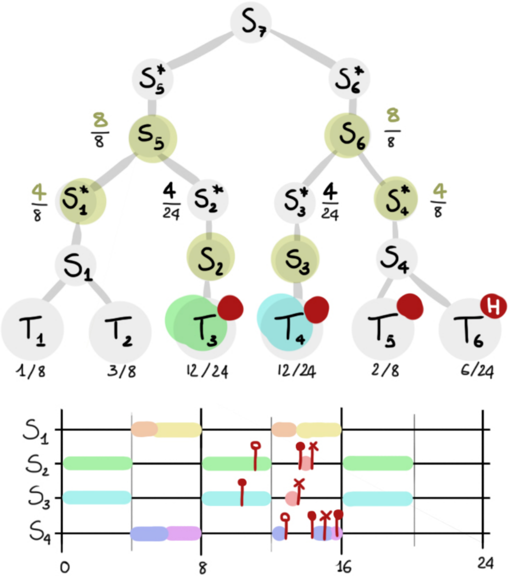
- spinning
- execution of critical section of tasks being helped

RUN+SBLP executing



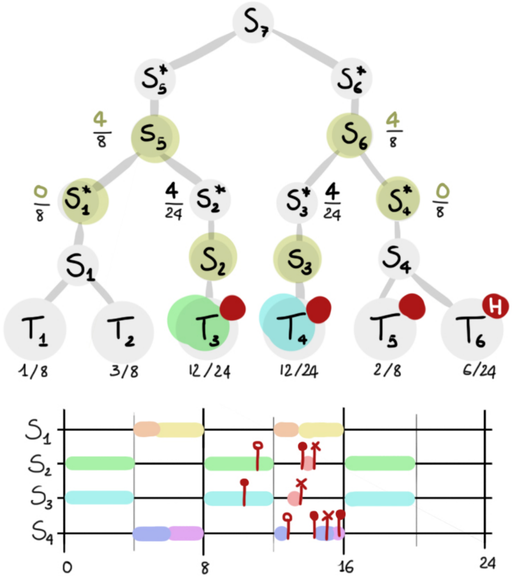
…T_5 uses the resource and completes before its deadline. Since the resource is free, T_6 can lock it.

RUN+SBLP executing



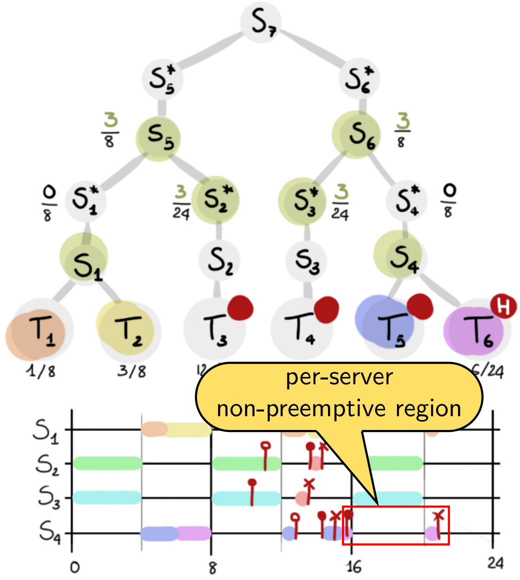
Then a scheduling decision of RUN lets S_2 and S_3 execute...

RUN+SBLP executing



…since T_3 and T_4 already used and released the resource, it's just normal execution and budget consumption.

RUN+SBLP executing

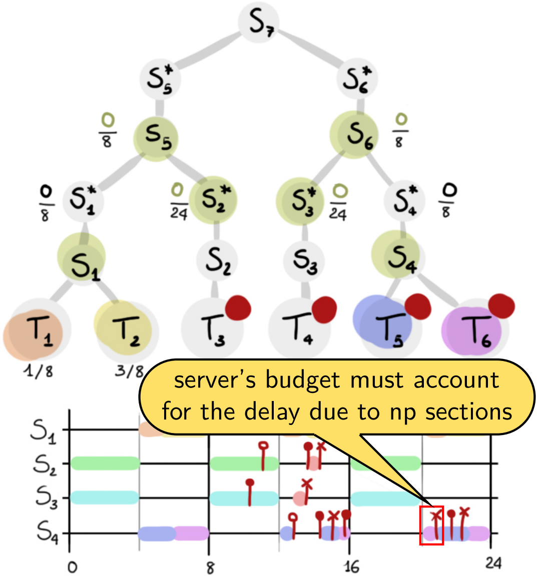


When S_4 must execute, it executes T_6 instead of T_5

- each request is a **per-server non-preemptive region**

Idea - collaborative tasks will use the same resource: let's avoid unneeded preemptions

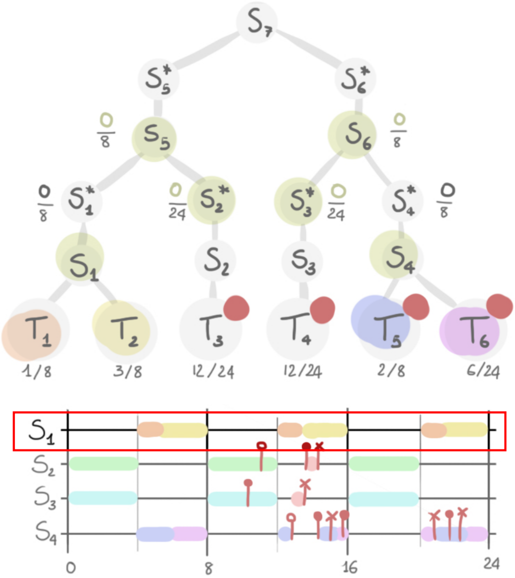
RUN+SBLP executing



...but the delay suffered from the tasks in the same server must be considered. To meet the deadlines of tasks

- **increase budget** of servers containing collaborative tasks

RUN+SBLP executing

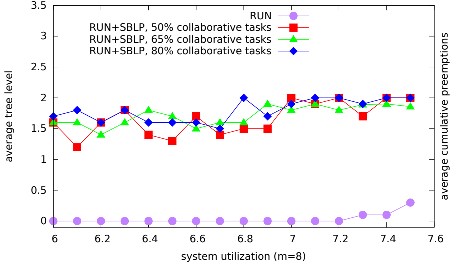


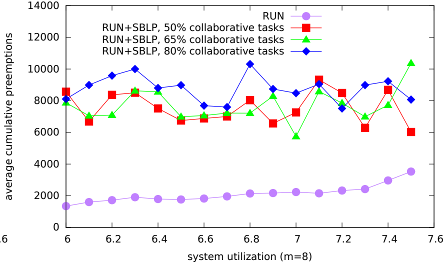
...and in the meanwhile, T_1 and T_2 (which were not using the resource) executed, completed and met their deadlines without any interference!

Implementation

SBLP implemented in the plugin of RUN for LITMUS^{RT}

- non-invasive for kernel primitives
- high impact on runtime preemptions and migrations
  - preemptions/migrations may be needed by the helping mechanism
  - height of reduction tree is increased ($\Rightarrow$ more preemptions/migrations)





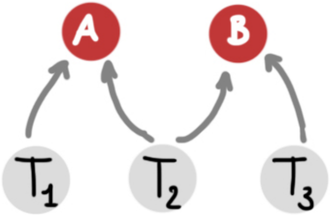
Conclusions

- \+ RUN can be used with resource-sharing tasks
- \+ Servers can be useful to overcome the limitations of partitioning while dealing with resources: reduced parallelism and ad-hoc packing
- − Increased runtime overhead

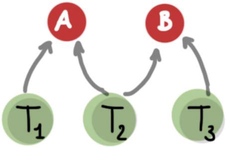
Multiple resources?

What happens if tasks use several shared resources?

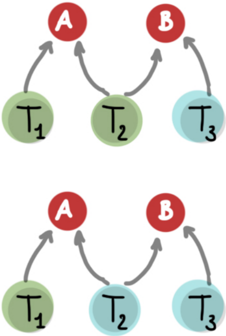
- protocol does not change
- highlights the limit of statically defined servers for collaborative tasks: decrease parallelism or decrease delay caused by unrelated resources?



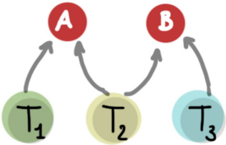
Grouping strategies



parallelism $= 1$
delay $=$ max



parallelism $= 1$ or 2
delay $=$ min or max



parallelism $= 2$
delay $=$ min

Parallelism and delays

System utilization is increased by two distinct amounts:

1. increased WCET of tasks: related to the number of parallel requests
2. increased budget of servers: related to the length of non-preemptive critical section

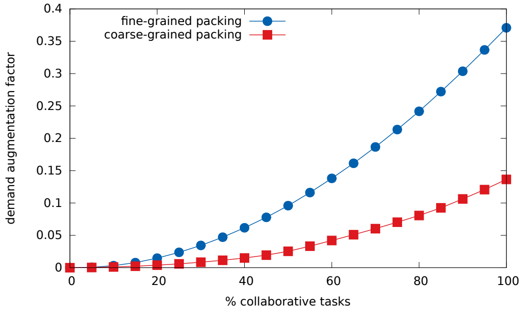
Which quantity is likely to affect the most the increased system utilization?

Schedulability simulations

Simulations using two packing heuristics for collaborative tasks

- coarse-grained: to reduce the length of FIFO queue of resources
- fine-grained: to avoid blocking caused by unrelated resources

$\Rightarrow$ *less servers is (generally) better!*



Nested resources?

Nested resources can be used. How to avoid deadlock?

- Ordered access
- Group lock
- Partial group lock: using group lock only for nesting tasks (or better, all nesting tasks in the same server)