Sporadic Server Revisited

Dario Faggioli, Marko Bertogna, Fabio Checconi

Scuola Superiore Sant’Anna, Pisa, Italy
Summary

• System Model
• Resource Reservation
• Original Sporadic Server
• Improved SS
  – Adaptations to multiprocessor systems
  – Overrun handling
  – Overhead reduction
• Simulations
• Conclusions
System Model

- Identical Multiprocessor Platforms
  - $m$ processors
- Hard, Soft and Non Real-Time tasks
  - Periodic and sporadic tasks: $(C_i, D_i, T_i)$
  - Aperiodic requests
- Fixed Priorities
Resource Reservation

- One or more tasks is assigned a Reservation (or server)
- Each reservation has a Budget $Q$ and a Period $P$
  - guaranteed to execute for $Q$ every $P$ time-units
- Bandwidth isolation among different tasks
- Aperiodic requests are served through a dedicated server
- Various solutions for FP servers
  - Polling Server (PS)
  - Deferrable Server (DS)
  - Sporadic Server (SS)
Existing Servers

- PS
  - ☒ Low responsiveness (budget discharged when no activity to serve)
  - ✔ Good schedulability

- DS
  - ✔ High responsiveness
  - ☒ Worse schedulability (back-to-back execution)

- SS
  - ✔ High responsiveness
  - ✔ Good schedulability
Sporadic Server [Sprunt et al.’89]

- When a server is activated, a recharge is posted
  - $P$ time-units after the activation
  - amount of capacity consumed since the activation

\[
SS (Q,P) \quad \text{recharge} \quad P \quad \text{recharge}
\]

\[
Q \quad +2 \quad P \quad +1
\]
Sporadic Server limitations

- Developed for single processor systems
- No schedulability test for multiprocessor systems
- No existing solution to handle server overruns
- Many recharging events during a server period
  - May lead to a large overhead

- Need further investigation to implement SS on a real OS
  (i.e., SCHED_SPORADIC on Linux)
SS for multiprocessors

• The workload of a SS in a window of length $L$ is maximized when SS is continuously backlogged

• Worst-case situation:

- SS behaves like a periodic task with WCET $Q$ and period $P$
- Existing tests for globally scheduled systems are applicable:
  - Utilization based test for RM-US[1/3]:
    \[ U_{\text{tot}} \leq (m+1)/3 \]
  - Response Time Analysis for FP systems (RTSS’07)
Server overrun

- In most cases it is not possible (or wanted) to enforce a budget of exactly $Q$ time-units for each SS
- Possible overrun causes:
  - Timer resolution
  - OS latency
  - Budget exhaustion during critical sections
Limited Timer Resolution

- Modern OS’s account for the elapsed execution time of a task/server in two ways:
  - Event-driven: at each scheduling event (i.e., task activation/termination)
  - Time-driven: periodically at each system tick (i.e., 1ms-10ms in Linux)

- The budget exhaustion might be detected with a delay of up to $P_{\text{tick}}$ (+ jitter)
Overrun due to OS latency

- Widened latencies might be due to particular configurations
  - i.e., virtualized systems: a guest OS is executed as a process of another (host) OS

- A tick of the guest system can be serviced only when the guest VM becomes a running process on the host
  - this might happen after a long time
  - the real resolution of the guest tick might significantly vary
Overrun due to critical sections

• When a server exhausts its budget $Q$ while still holding a lock on a shared resource, it is not wise to preempt it:
  – Preempting tasks might be blocked on that resource, without making any progress to the system

• For PS and DS, two possible solutions have been studied:
  – Budget check before each locking operation [Fisher et al.’07, Bertogna et al.’09]
  – Overrun and payback [Davis et al.’06, Behnam et al.’07]

• First solution requires the OS to know the length of each critical section → more suitable for hard RTOS’s

• Second solution more appropriate for general purpose OS’s, but needs to handle overruns
Overrun and payback

- When overruns cannot be avoided, a payback mechanism is needed to avoid over-provisioning.
- For PS and DS, the overrun amount is simply paid back by the server during its subsequent instance(s).

\[
(Q + O_1) - O_1 = Q
\]

Permanent capacity loss!

- The same technique does not work for SS.

\[
(Q + O_2) - O_1 - O_2
\]

Permanent capacity loss!
Solution

- The overhead amount $O$ is stored in an overrun pool $O_{\text{pool}}$:
  
  $O_{\text{pool}} + O$

- Each recharge amount is limited to at most the server's budget $Q$

- When a replenishment occurs:
  - The server budget $q$ is updated (increased)
  - Both $O_{\text{pool}}$ and $q$ are decreased by $\min(q, O_{\text{pool}})$
  - A recharge of $\min(q, O_{\text{pool}})$ is set after $P$ time-units

- Consider the overhead as it is consumed at the next replenishment time

$O_1$ instantly consumed here
$O_2$ instantly consumed here
Schedulability with overruns

- **Without payback** → each SS with overrun $\Delta$ is accounted a $WCET = Q + \Delta$
- **With payback** → Improved RTA formulation
  - Theorem: In any interval $L$, the cumulative execution of SS cannot exceed by more than $\Delta$ time-units the one of a non-overrunning server with identical budget and period
Simulation results

- Randomly generated load of sporadic servers
  - 10^6 sets of servers
  - P uniformly distributed in [10^4-10^6]
  - Q = U·P, where U follows a normal distribution with mean 0.25

- Percentage of sets that are found schedulable only with our improved payback method:

<table>
<thead>
<tr>
<th></th>
<th>P_{tick} = 1000</th>
<th>P_{tick} = 4000</th>
<th>P_{tick} = 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 2</td>
<td>44%</td>
<td>45%</td>
<td>47%</td>
</tr>
<tr>
<td>m = 4</td>
<td>48%</td>
<td>49%</td>
<td>51%</td>
</tr>
<tr>
<td>m = 8</td>
<td>53%</td>
<td>54%</td>
<td>55%</td>
</tr>
</tbody>
</table>
Useless replenishing events

• Bursty activities might lead to many replenishing events:

• Overhead associated to these events
  – Running process is interrupted to run the timer handling routine
  – If system was idle, exit from idle state

• Solution: one cumulative budget replenishment
  – Stop timers when server is not backlogged
  – When server activates again, check for elapsed replenishing events and rearm timers for future events
Improvement evaluation

- Each server is activated with a certain probability
- At each activation it executes $K$ sub-instances (for $Q/K$ units)
- The number of wakeups avoided by our technique (normalized over 1000 time-units) is:

![Graph showing wakeups vs. skip probability for different values of $K$ and burstiness levels]
Conclusions

• Refined Sporadic Server definition
• Focus on implementation considerations
  – In particular, SCHED_SPORADIC class on Linux
• Extension to globally scheduled multiprocessor systems
• Efficient payback mechanism in overrun situations
• Reduced overhead due to timer handling routines
• Improved schedulability with a small implementation overhead
SS schedulability on Xproc

• Possible to apply existing tests for globally scheduled sporadic task systems
  – Utilization based test for RM-US[1/3]:
    \[ U_{tot} \leq \frac{(m+1)}{3} \]
  – Response Time Analysis for FP systems (RTSS’07)

\[
W_i(L, s_i) = \min (C_i, L + D_i - C_i - s_i - N_i(L, s_i)T_i) \]
\[ + N_i(L, s_i)C_i, \]
\[
N_i(L, s_i) = \left\lfloor \frac{L + D_i - C_i - s_i}{T_i} \right\rfloor \]
\[
s_i = D_i - C_i - \left\lfloor \frac{\sum_{j<i} \min(W_j(D_i, s_j), D_i - C_i + 1)}{m} \right\rfloor \]