Scalable Memory Reclamation for Multi-Core, Real-Time Systems

**Yuxin Ren**, Guyue Liu, Gabriel Parmer
The George Washington University

Björn Brandenburg
Max Planck Institute for Software Systems
Background: Multicore, Real Time System

How to efficiently share resources?
Background: Multicore, Real Time System

How to provide real-time guarantee?
Shared Linked List Example

Core\textsubscript{1} \quad \textit{read(b)} \quad \textit{read(c)} \quad \textit{update(b)}

\quad \text{a} \quad \text{b} \quad \text{c}
Linked List Example: Global Lock

Core_1 \rightarrow \text{read(b)} \rightarrow \text{Core_2} \rightarrow \text{read(c)} \rightarrow \text{Core_3} \rightarrow \text{update(b)}

Linked List:

a \rightarrow b \rightarrow c
Global Lock: blocking
Global Lock: blocking

\[ T_1 \]

\[ T_2 \]

\[ \text{time} \]
Global Lock: blocking
Linked List Example: Fine-grained Lock

Core₁ \( \rightarrow \) Core₂ \( \rightarrow \) Core₃

read(b) \( \rightarrow \) read(c) \( \rightarrow \) update(b)

a \( \rightarrow \) b \( \rightarrow \) c
Locking: scalability

99th percentile measured cost

- Ticket lock
- MCS lock

Intel Xeon E7-4850
4 sockets, 40 cores
Linked List Example:
Read Copy Update (RCU)
Scalable Memory Reclamation (SMR)
Linked List Example: RCU/SMR

![Diagram of linked list with cores and update/read operations]
Linked List Example: RCU/SMR

Core$_1$ \quad Core$_2$ \quad Core$_3$

read(b) \quad update(b)

a \quad b \quad c
Linked List Example: RCU/SMR
Linked List Example: RCU/SMR

Parallel Readers
Reduced response time

\( T_1 \)
\( T_2 \)
\( T_3 \)
Scalable Memory Reclamation (SMR)

update(b)

b' = alloc()
Scalable Memory Reclamation (SMR)

$\text{update}(b)$

$\text{b}' = \text{alloc}()$

$\text{free}(b)$
Scalable Memory Reclamation (SMR)

\[ b' = alloc() \text{ free}(b) \]

Writer (Core_3)

Reader (Core_2)

Reader (Core_1)

\[ t_1 \quad t_2 \]
Scalable Memory Reclamation (SMR)

Writer (Core₃)

Reader (Core₂)

Reader (Core₁)

\[ b' = \text{alloc()} \ \text{free}(b) \]

\[ b = \text{alloc()} \]

\[ b \]

\[ t_1 \]

\[ t_2 \]

\[ t_3 \]
Scalable Memory Reclamation (SMR)

$\text{Writer (Core}_3\text{)}$

$\text{Reader (Core}_2\text{)}$

$\text{Reader (Core}_1\text{)}$

$b' = \text{alloc()} \ free(b)$

$c = \text{alloc()}$

$t_1, t_2, t_3$
Scalable Memory Reclamation (SMR)

Writer (Core$_3$)

Reader (Core$_2$)

Reader (Core$_1$)

b’ = alloc() free(b)

c = alloc()

b = alloc()

t$_1$  t$_2$  t$_3$  t$_4$
Scalable Memory Reclamation (SMR)

When to reuse freed memory?

Writer
(Core₃)

Reader
(Core₂)

Reader
(Core₁)

b' = alloc() free(b)

b = alloc()

c = alloc()

When to reuse freed memory?
Scalable Memory Reclamation (SMR)

Scalability Problem $\rightarrow$ Garbage Collection Problem
Scalable Memory Reclamation (SMR)

When to reuse freed memory? Quiescence state

\[ b' = \text{alloc()} \ \text{free}(b) \]
\[ c = \text{alloc()} \]
Scalable Memory Reclamation (SMR)

When to reuse freed memory? Quiescence state

- Writer (Core₃)
- Reader (Core₂)
- Reader (Core₁)

$t₁$, $t₂$, $t₃$
Scalable Memory Reclamation (SMR)

When to reuse freed memory? Quiescence state
Scalable Memory Reclamation (SMR)

When to reuse freed memory? Quiescence state → Grace period
SMR: API

- **enter**: declare the start of code referencing a shared data structure
- **exit**: declare the end of code referencing that data structure

- **sync**: calculate grace period
  - wait till memory is reusable
  - delay the reuse of memory

```c
enter();
n = walk(path);
process(n);
exit();
// no references into DS remain
```

```c
enter();
n = unlink(path);
exit();
sync();
// reuse prior memory
```

---

Read-path

Update-path
SMR Implementation: U-RCU example

**sync**: suspend, wait for the grace period
Blocking increases the worst case response time.
SMR Implementation: U-RCU example

**Sync:** suspend, wait for the grace period

**Priority Inversion**
SMR Implementation: U-RCU example

Writer (Core_3)

Reader (Core_2)

Reader (Core_1)

Priority Inversion

Self-suspension

sync: suspend, wait for the grace period

t_1

t_2

U-RCU

free

sync

Blocking

enter

exit

sync: suspend, wait for the grace period
# Summary of Existing Solutions

<table>
<thead>
<tr>
<th></th>
<th>Predictability</th>
<th>Scalability</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>U-RCU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real-time SMR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Summary of Existing Solutions

<table>
<thead>
<tr>
<th></th>
<th>Predictability</th>
<th>Scalability</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>U-RCU</td>
<td>✗</td>
<td>✔</td>
<td>○</td>
</tr>
<tr>
<td>Real-time SMR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Summary of Existing Solutions

<table>
<thead>
<tr>
<th></th>
<th>Predictability</th>
<th>Scalability</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>✅</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>U-RCU</td>
<td>✗</td>
<td>✅</td>
<td>○</td>
</tr>
<tr>
<td>Real-time</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>SMR</td>
<td>✅</td>
<td>✅</td>
<td></td>
</tr>
</tbody>
</table>
Real-Time SMR

• Two Implementations:
  • RT-ParSec: scalable bounded memory consumption
  • Temporal Quiescence: optimized for hard real time system

• Analytical Model:
  • Worst case memory consumption.
  • Response Time Analysis of quiescence calculations.
Real-Time SMR: RT-ParSec

Writer (Core$_3$)

Reader (Core$_2$)

Reader (Core$_1$)

\[ \text{free} \]

\[ \text{free} \]

\[ \text{sync} \]

\[ t_0 \]

\[ t_1 \]

\[ t_2 \]

\[ \text{time} \]
Real-Time SMR: RT-ParSec

Writer (Core₃)

Reader (Core₂)

Reader (Core₁)

Grace period
Real-Time SMR: RT-ParSec

Writer (Core₃)

Reader (Core₂)

Reader (Core₁)

Grace period

Time

t₀  t₁  t₂
Real-Time SMR: RT-ParSec

![Diagram of Real-Time SMR: RT-ParSec]

- **Writer (Core₃)**: free, free, free, free, sync
- **Reader (Core₂)**: free
- **Reader (Core₁)**: free

- **Grace period**: $t_0$ to $t_1$
- **U-RCU** blocking period: $t_1$ to $t_2$

- **Time points**: $t_0$, $t_1$, $t_2$, $t_3$
Real-Time SMR: RT-ParSec

Writer (Core₃)

Reader (Core₂)

Reader (Core₁)

free  free  free  free  sync

t₀  t₁  t₂  t₃

Blocking

RT-ParSec  Grace period  Grace period  U-RCU

Writer (Core₃) is initially free and then blocks at time t₁ due to a sync event. Readers (Core₂ and Core₁) read from the memory during their respective Grace periods (t₂ and t₃).
Real-Time SMR: Temporal Quiescence

Writer (Core$_3$)

Reader (Core$_2$)

Reader (Core$_1$)

time

t$_0$
t$_1$
t$_2$

free

free

sync
Real-Time SMR: Temporal Quiescence

Writer (Core\textsubscript{3})

Reader (Core\textsubscript{2})

Reader (Core\textsubscript{1})

Free
Free
Free
Free
Sync

\[ t_0 \leq t_1 \leq t_2 \]

Temporal

\[ \leq \text{Longest Grace Period} \]
Real-Time SMR: Temporal Quiescence

Perfectly scalable
no shared cache lines
System Model and Assumptions

- Sporadic task model
- Partitioned fixed priority scheduling
- Non nested requests
- Memory reclamation as quiescence detection task
Worst Case Memory Consumption

\[ A(W) = \sum_{i=1}^{N} \Delta(W) \times a_i \]
Worst Case Memory Consumption

\[ A(W) = \sum_{i=1}^{N} \Delta(W) \times a_i \]

- Number of writers
- Max number of memory allocations
Worst Case Memory Consumption

\[ A(W) = \sum_{i=1}^{N} \Delta(W) \times a_i \]

Number of writers

Max number of memory allocations

Time interval where memory accumulates
Worst Case Memory Consumption

- **alloc**
- **free**
- **sync**

- Live
- Zombie
- Garbage
- Free

- Max allocation deallocation interval
- Longest grace period
- Quiescence detection period
- Quiescence detection task response time

- Alloc
- Free
- Sync
Response Time Analysis

\[ r_i = e_i + b_i + \sum_{\tau_k \in h_p} \left[ \frac{r_i}{p_k} \right] \times e_k \]
Response Time Analysis

Task execution time

System overhead

Worst case execution time

\[ r_i = e_i + b_i + \sum_{\tau_k \in hp_i} \left\lfloor \frac{r_i}{p_k} \right\rfloor \times e_k \]

Interference

Higher priority tasks

Quiescence detection tasks

50
Response Time Analysis

\[ r_i = e_i + b_i + \sum_{\tau_k \in hp_i} \left\lfloor \frac{r_i}{p_k} \right\rfloor \times e_k \]

- Task execution time
- System overhead
- Worst case execution time
- Interference
- Total blocking time (U-RCU)
- Higher priority tasks
- Quiescence detection tasks
Interdependency

Memory accumulation → Quiescence task response time

↑

↑
Interdependency

Memory accumulation $\iff$ Quiescence task response time
Evaluation: Schedulability Tests
Evaluation: Schedulability Tests

![Graph showing the relationship between the number of cores and schedulability for different lock mechanisms: RT-ParSec, Temporal, MCS lock w/ ILP, phase-fair RW lock, U-RCU.](image)

- **RT-ParSec**
- **Temporal**
- **MCS lock w/ ILP**
- **Phase-fair RW lock**
- **U-RCU**
Evaluation: Memcached

Memcached Throughput

Million Requests Per Second

Number of Cores

RT-ParSec

phase-fair RW lock

MCS lock
Evaluation: Memcached

Graph showing the Memcached Throughput vs. Number of Cores. The x-axis represents the number of cores, ranging from 5 to 40, and the y-axis represents Million Requests Per Second, ranging from 5 to 35. The graph compares different lock mechanisms:

- **RT-ParSec** (green line with crosses)
- **phase-fair RW lock** (blue line with circles)
- **MCS lock** (red line with triangles)

The graph indicates that RT-ParSec consistently outperforms the other two mechanisms, with a notable improvement around 35 cores, showing a 1.73x increase in throughput compared to MCS lock.
Evaluation: Memcached

99th Percentile Get Latency

- MCS lock
- pf-RW lock
- RT-ParSec

Get Request (µs)

Number of Cores

Number of Cores: 5, 10, 15, 20, 25, 30, 35, 40

Get Request (µs): 0, 5, 10, 15, 20, 25, 30, 35, 40
Evaluation: Memcached

![Graph showing 99th percentile get latency for different number of cores and lock types: MCS lock, pf-RW lock, RT-ParSec. The graph indicates a 6.89x improvement in latency for RT-ParSec compared to MCS lock at 40 cores.](image-url)
Conclusion

Real-time SMR: Scalable predictable resource sharing

• Scalable and predictable quiescence detection

• Bounds on memory utilization

• Bounds on response time
https://github.com/gwsystems/ps