Eliminating Read Barriers through Procrastination and Cleanliness

KC Sivaramakrishnan
Lukasz Ziarek
Suresh Jagannathan
Lightweight user-level threads

Scheduler 1

Lots of concurrency!
Big Picture

Expendable resource?

Scheduler 1

Lots of concurrency!

Heap
Exploit program concurrency
to
eliminate read barriers from thread-local collectors

Expendable resource?

Lots of concurrency!

Alleviate MM cost?
MultiMLton

• Goals
  – Safety, Scalability, ready for future manycore processors
• Parallel extension of MLton – a whole-program, optimizing SML compiler
• Parallel extension of Concurrent ML
  – Lots of Concurrency!
  – Interact by sending messages over first-class channels

\[ \text{send} (c, v) \]
\[ v \leftarrow \text{recv} (c) \]
MultiMLton GC: Considerations

• Standard ML – functional PL with side-effects
  – Most objects are small and ephemeral
    • Independent generational GC
  – # Mutations << # Reads
    • Keep cost of reads to be low

• Minimize NUMA effects

• Run on non-cache coherent HW
MultiMLton GC: Design

- NUMA Awareness
- Circumvent cache-coherence issues
Invariant Preservation

- Read and write barriers for preserving invariants

```
Target

Shared Heap
r

Local Heap
x

r := x

Source

Exporting writes

Mutator needs read barriers!

Shared Heap
r

x

FWD

Local Heap

Transitive closure of x
```
Challenge

- Object reads are pervasive
  - RB overhead \( \propto \) cost (RB) * frequency (RB)
- Read barrier optimization
  - Stacks and Registers never point to forwarded objects

![Bar Graph](image-url)

- Mean Overhead:
  - AMD: 20.1%
  - SCC: 15.3%
  - AZUL: 21.3%
Mutator and Forwarded Objects

\[
\frac{\text{# Encountered forwarded objects}}{\text{# RB invocations}} < 0.00001
\]

Eliminate read barriers altogether
RB Elimination

• Visibility Invariant
  – Mutator does not encounter forwarded objects

• Observation
  – No forwarded objects created $\Rightarrow$ visibility invariant $\Rightarrow$ No read barriers

• Exploit concurrency $\Rightarrow$ Procrastination!
Procrastination

\[ \rightarrow r_1 := x_1 \]

\[ \rightarrow r_2 := x_2 \]

\[ \rightarrow T \text{ is running} \]

\[ \rightarrow T \text{ is suspended} \]

\[ \rightarrow T \text{ is blocked} \]
Procrastination

r1 := x1
→ r2 := x2

Control switches to T2

T is running

T is suspended

T is blocked

Delayed write list
Procrastination

\[ r_1 := x_1 \quad r_2 := x_2 \]

- \( T \) is running
- \( T \) is suspended
- \( T \) is blocked

Delayed write list

\[ \rightarrow \]

Shared Heap

Local Heap

\[ r_1 \quad r_2 \]

\[ x_1 \quad x_2 \]
Procrastination

T1

T2

r1 := x1

r2 := x2

T → T is running

T → T is suspended

T → T is blocked

Delayed write list → 

T → FWD

T → FWD

Shared Heap

r1

x1

r2

x2

Local Heap
Procrastination

\[ \rightarrow r_1 := x_1 \]

\[ \rightarrow r_2 := x_2 \]

\[ \rightarrow T \text{ is running} \]

\[ \rightarrow T \text{ is suspended} \]

\[ \rightarrow T \text{ is blocked} \]

\[ \text{Force local GC} \]

Delayed write list \[ \rightarrow \]

\[ \text{Shared Heap} \]

\[ r_1 \rightarrow x_1 \rightarrow r_2 \rightarrow x_2 \]

\[ \text{Local Heap} \]
Correctness

• Does Procrastination introduce deadlocks?
  – Threads can be procrastinated while holding a lock!

\[
\begin{align*}
\text{T1} & \quad \text{T2} \\
\rightarrow \quad & \text{is running} \\
\rightarrow \quad & \text{is suspended} \\
\rightarrow & \text{is blocked}
\end{align*}
\]
Correctness

• Does Procrastination introduce deadlocks?
  – Threads can be procrastinated while holding a lock!

• Is Procrastination safe?
  – Yes. Forcing a local GC unblocks the threads.
  – No deadlocks or livelocks!
Correctness

• Does Procrastination introduce deadlocks?
  – Threads can be procrastinated while holding a lock!

• Is Procrastination safe?
  – Yes. Forcing a local GC unblocks the threads.
  – No deadlocks or livelocks!
Is Procrastination alone enough?

- Efficacy (Procrastination) $\propto$ # Available runnable threads

- With Procrastination, half of local major GCs were forced

  Eager exporting writes while preserving visibility invariant
Cleanliness

• A clean object closure can be lifted to the shared heap without breaking the visibility invariant

\[
\text{r := x} \quad \text{inSharedHeap (r)} \\
\text{inLocalHeap (x)} \quad \text{&&} \\
\text{isClean (x)} \\
\]

Eager write (no Procrastination)
Cleanliness: Intuition

Shared Heap

Local Heap

lift \( (x) \) to shared heap
Cleanliness: Intuition

find all references to FWD
Cleanliness: Intuition

Need to scan the entire local heap

![Diagram showing shared heap and local heap with a node x connecting them.]
Cleanliness: Simpler question

Do all references originate from heap region $h$?

```
sizeof (h) << sizeof (local heap)
```
Cleanliness: Simpler question

sizeof (h) << sizeof (local heap)
Heap Sessions

• Source of an exporting write is often
  – Young
  – rarely referenced from outside the closure

• Current session closed & new session opened
  – After an exporting write, a user-level context switch, a local GC
Heap Sessions

- Source of an exporting write is often
  - Young
  - rarely referenced from outside the closure

- Current session closed & new session opened
  - After an exporting write, a user-level context switch, a local GC
  - SessionStart is moved to Frontier

- Average current session size \(< 4\text{KB}\)
Cleanliness: Eager exporting writes

- A clean object closure
  - is **fully contained** within the current session
  - has **no references** from previous session

![Diagram showing previous and current sessions with clean object closure]

- `r := x`
Cleanliness: Eager exporting writes

- A clean object closure
  - is **fully contained** within the current session
  - has **no references** from previous session

\[ r := x \]
Avoid tracing current session?

• Many SML objects are tree-structured (List, Tree, etc.,)
  – Specialize for no pointers from outside the object closure

• $\forall x' \in$ transitive object closure ($x$),
  \[
  \text{ref}_\text{count} (x) = 0 \&\& \text{ref}_\text{count} (x') = 1
  \]
  – ref\_count does not consider pointers from stack or registers

• Eager exporting write
  – No current session tracing needed!
Reference Count

• Purpose
  – Track pointers from previous session to current session
  – Identify tree-structured object

• Does not track pointers from stack and registers
  – Reference count only triggered during object initialization and mutation
Bringing it all together

• $\forall x' \in \text{transitive object closure (x)},$

  \[
  \text{if } \max(\text{ref\_count (x'))}
  \]

  – One & ref\_count (x) = 0 $\Rightarrow$ tree-structured (Clean)
    $\Rightarrow$ Session tracing not needed
  – LocalMany $\Rightarrow$ Clean $\Rightarrow$ Trace current session
  – Global $\Rightarrow$ 1+ pointer from previous session $\Rightarrow$
    Procrastinate
Example 1: Tree-structured Object

Local Heap

Previous Session

current stack

T1

r := x

Shared heap
Example 1: Tree-structured Object

Local Heap

Previous Session

Local Heap

current stack

T1

FWD

Walk current stack

Current Session

r := x

Shared heap

r

x

z

y
Example 1: Tree-structured Object

Local Heap

Previous Session

Current Session

T1

No need to walk current session!

r := x

Shared heap

Local Heap
r
x
z
y

Previous Session

Local Heap

Current Stack

T1

No need to walk current session!
Example 1: Tree-structured Object

Local Heap

Previous Session

R := x

Shared heap

Current Session

T1

FWD

T2

Next stack
Example 1: Tree-structured Object

Local Heap

Previous Session

Local Heap

previous stack

T1

Context Switch

Current Session

T2

current stack

Shared heap

T1

\[ r := x \]

T2

Walk target stack

Shared heap

Previous Session

Local Heap

Previous stack

T1

Context Switch

Current Session

T2

current stack

Shared heap

T1

\[ r := x \]

T2

Walk target stack

Shared heap
Example 2: Object Graph

Local Heap

Previous Session

Current stack

Current Session

r := x

r

Shared heap
Example 2: Object Graph

r := x

Shared heap

Local Heap

Previous Session

current stack

Current Session

Walk current stack

Walk current session

FWD

FWD

a

40
Example 2: Object Graph

Local Heap

Previous Session

Current Session

Shared heap

current stack

Walk current stack

Walk current session

r := x

r := x

a

r

x

z

y

Shared heap
Example 3: Global Reference

```
Local Heap

Previous Session

r := x

Current Session

T1

current stack

Shared heap

```
Example 3: Global Reference

Local Heap

Previous Session

Shared heap

Current Session

a

z(G)

x(0)

y(1)

T1

current stack

r := x

Procrastinate

r
Immutable Objects

• Specialize exporting writes

• If immutable object in previous session
  – Copy to shared heap
    • Immutable objects in SML do not have identity
  – Original object unmodified

• Avoid space leaks
  – Treat large immutable objects as mutable
Cleanliness: Summary

• Cleanliness allows eager exporting writes while preserving visibility invariant

• With Procrastination + Cleanliness, $<1\%$ of local GCs were *forced*
Evaluation

• Variants
  – **RB-**: TLC with Procrastination and Cleanliness
  – **RB+**: TLC with read barriers

• Sansom’s dual-mode GC
  – Cheney’s 2-space copying collection \( \leftrightarrow \) Jonker’s sliding mark-compacting
  – Generational, 2 generations, No aging

• Target Architectures:
  – 16-core AMD Opteron server (NUMA)
  – 48-core Intel SCC (non-cache coherent)
  – 864-core Azul Vega3
Results

• **Speedup:** At 3X min heap size, RB- faster than RB+
  – AMD (16-cores) **32%** (2X faster than STW collector)
  – SCC (48-cores) **20%**
  – AZUL (864-cores) **30%**

• **Concurrency**
  – During exporting write, **8** runnable user-level threads/core!
Cleanliness Impact

- **RB- MU-**: RB- GC ignoring mutability for Cleanliness
- **RB- CL-**: RB- GC ignoring Cleanliness (*Only Procrastination*)
Conclusion

• Eliminate the need for read barriers by preserving the visibility invariant
  – **Procrastination**: Exploit concurrency for delaying exporting writes
  – **Cleanliness**: Exploit generational property for eagerly perform exporting writes

• Additional niceties
  – Completely dynamic \(\rightarrow\) Portable
  – Does not impose any restriction on the GC strategy
Questions?

http://multimlton.cs.purdue.edu
Results

• On AMD, 16 Cores, 3X minimum heap size

• **Mutator time:**
  – STW GC spends the least amount of time in the mutator
    • No read/write barriers
  – Compared to STW GC, the mutator time of
    • RB- 18% more, RB+ 39% more

• **GC time:**
  – RB- spends the least amount of time doing GC
  – RB- within 5% of RB+
At 3X min heap size:

- RB+: 32%
- STW: 106%
- BDW: 584%

Figure 8.8+ Performance comparison of Stop4the4world (STW) Boehm-Demers-Weiser conservative garbage collector (BDW) local collector

As we decrease the heap sizes on SCC, compared to the fastest time graphs, we can see that the programs tend to run much slower.

Only consider performance of our local collectors since our AMD (16-cores) machine with programs running on 8 cores with the benchmark.

Cleanliness information allows the runtime system to avoid pre-copying collections, which is noticeable by a more rapid increase in garbage collection overhead due to the increased number of shared heap collections, which are due to the increased number of shared heap collections.
Programming Models for the Intel SCC Many-core Processor Workshop APMM 2011 as Part of the HPCS 2011

- Strictly No Cache Coherency
- Cluster-on-Chip Architecture
- Private off-die DRAM Regions (one per Core)
  - Caches enabled
  - One Linux instance per Core!
- Shared / Global off-die DRAM Region
  - Caches disabled per default!
  - e.g. for global shared data
- Shared on-die MPB Regions
  - Cached in L1, L2 Bypass / Fast L1 Invalidation for MPB-Data

Shared off-die DRAM

Private DRAM L2$ L1$ CPU0

Private DRAM L2$ L1$ CPU47

Message Passing Buffer (8KB/core)
Total time: SCC and AZUL

SCC (48-cores)

AZUL (864-Cores)

Non-cache coherent

Scalable, cache-coherent
Cleanliness Impact (1)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AllPairs</th>
<th>BarnesHut</th>
<th>CountGraphs</th>
<th>GameOfLife</th>
<th>Kclustering</th>
<th>Mandelbrot</th>
<th>Nucleic</th>
<th>Raytrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB-</td>
<td>1831</td>
<td>46532</td>
<td>154</td>
<td>38621</td>
<td>25812</td>
<td>132</td>
<td>156</td>
<td>3523</td>
</tr>
<tr>
<td>RB- MU-</td>
<td>1831</td>
<td>409232</td>
<td>192</td>
<td>735543</td>
<td>50323</td>
<td>209</td>
<td>433092</td>
<td>3743</td>
</tr>
<tr>
<td>RB- CL-</td>
<td>124232</td>
<td>67156821</td>
<td>50178</td>
<td>5867423</td>
<td>27023911</td>
<td>25491</td>
<td>912349</td>
<td>61198</td>
</tr>
</tbody>
</table>

Number of Preemptions on exporting writes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AllPairs</th>
<th>BarnesHut</th>
<th>CountGraphs</th>
<th>GameOfLife</th>
<th>Kclustering</th>
<th>Mandelbrot</th>
<th>Nucleic</th>
<th>Raytrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB-</td>
<td>0.08</td>
<td>0.17</td>
<td>0</td>
<td>3.54</td>
<td>0</td>
<td>1.43</td>
<td>0</td>
<td>1.72</td>
</tr>
<tr>
<td>RB- MU-</td>
<td>0.08</td>
<td>19.2</td>
<td>0.03</td>
<td>9.47</td>
<td>0.02</td>
<td>2.86</td>
<td>9.37</td>
<td>1.72</td>
</tr>
<tr>
<td>RB- CL-</td>
<td>38.55</td>
<td>100</td>
<td>0.18</td>
<td>99.75</td>
<td>21.64</td>
<td>86.22</td>
<td>19.3</td>
<td>24.86</td>
</tr>
</tbody>
</table>

Forced GCs as a % of total number of local major GCs
## Benchmark Characteristics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Allocation Rate (MB/s)</th>
<th>Bytes Allocated (GB)</th>
<th># Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AMD</td>
<td>SCC</td>
<td>AZUL</td>
</tr>
<tr>
<td>AllPairs</td>
<td>817</td>
<td>53</td>
<td>1505</td>
</tr>
<tr>
<td>Barneshut</td>
<td>772</td>
<td>70</td>
<td>1382</td>
</tr>
<tr>
<td>Countgraphs</td>
<td>2594</td>
<td>144</td>
<td>4475</td>
</tr>
<tr>
<td>GameOfLife</td>
<td>2445</td>
<td>127</td>
<td>4266</td>
</tr>
<tr>
<td>Kclustering</td>
<td>3643</td>
<td>108</td>
<td>8927</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>349</td>
<td>43</td>
<td>669</td>
</tr>
<tr>
<td>Nucleic</td>
<td>1430</td>
<td>87</td>
<td>4761</td>
</tr>
<tr>
<td>Raytrace</td>
<td>809</td>
<td>54</td>
<td>2133</td>
</tr>
</tbody>
</table>
To do this, we ignore the test for mutability in the cleanliness check of our local collector design with mutability information in mind. Hence, we study the ability to distinguish between mutable and immutable objects in new shared heap object to the target. Hence, we can ignore the copy of the object in the shared heap and assign a reference to the object.

If the source of an exporting write is immutable, we can make a new session to fix any references to forwarded objects. Without this, the source object mutability does not play a significant factor. For benchmarks such as GameOfLife, it is less than 9%.

On average, programs tend to run ~1% slower if cleanliness information is ignored. The results show that cleanliness analysis therefore plays a significant role in our GC design. Cleanliness information is ignored. The results show that cleanliness optimizes programs such as GameOfLife, which is due to forced GCs if cleanliness information is not used, whereas local collector designs with all of the features enabled; RB row MU row in Figure 9, and Figure 9; show the number of preemptions on write barrier for different local collector configurations. RB row represents the local major collection performed.

Figure 9: Impact of heap session: % LM clean represents the fraction of instances where the source of an exporting write is clean with at least one of the objects in the closure has a source object mutability of LOCAL MANY references.

### Session Impact

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AllPairs</th>
<th>BarnesHut</th>
<th>Countgraphs</th>
<th>GameOfLife</th>
<th>Kclustering</th>
<th>Mandelbrot</th>
<th>Nucleic</th>
<th>Raytrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>% LM clean</td>
<td>5.3</td>
<td>13.4</td>
<td>8.6</td>
<td>23.2</td>
<td>17.6</td>
<td>4.5</td>
<td>13.3</td>
<td>8.2</td>
</tr>
<tr>
<td>Avg. session size (bytes)</td>
<td>2908</td>
<td>1580</td>
<td>3612</td>
<td>1344</td>
<td>2318</td>
<td>8723</td>
<td>1264</td>
<td>1123</td>
</tr>
</tbody>
</table>
Read Barrier

Conditional (Baker Style)

From

To

Unconditional (Brooks style)

From

To

PURDUE UNIVERSITY
pointer readBarrier (pointer *p) {
    if (*((Header*)(p - HD_OFF)) == F)
        return *(pointer*)p;
    return p;
}

pointer readBarrier (pointer *p) {
    return *(pointer*)(p - IND_OFF);
}

Has Conditional Check

Needs extra header word
Read Barrier Optimizations

- Stacks and registers never point to forwarding pointers
- “Eager” read barriers (D. Bacon et al. POPL’93)
- Scan stack after exporting write
- Exporting write is a GC safe-point
- Reduces RB overhead by ~5%
Performance on AZUL

At 3X min heap size:

- RB+ 30%
Performance on SCC

At 3X min heap size:

RB+ 20%

(a) Total time

(b) Mutator time

(c) Garbage collection time

(d) Garbage collection overhead
Under the hood

T1

send (c, v)

v \leftarrow \text{recv} (c)

T2

Abstract Shared Heap

T1’s local heap

T2’s local heap

Before Communication
Under the hood

send \((c, v)\)

\(v \leftarrow \text{recv} \,(c)\)

Abstract Shared Heap

After Communication