

GC II:
The Final Outrage

Today's cool hacks

- Heap transactions, atomic fission & super-fast allocation
- Write barriers & generational GC
- Read barriers & real time / concurrent GC

A transactional view of heap management

Heap is in inconsistent state during the operation, so basic heap operations must be *atomic*:

- allocate
- collect
- mutate

All GC'd systems that provide concurrency (including simple interrupts) must handle this problem.

Techniques are relevant to Java, ML, Haskell, Scheme, Lisp, Smalltalk, . . .

Allocating

```
;; r1 := cons(r2,r3)  
;; fp is frontier pointer
```

```
r1 := fp
```

```
fp += 8
```

```
r1[0] := r2
```

```
r1[4] := r3
```

General strategies

Locking overhead must be very cheap:

- Consing is ubiquitous.
- Consing is cheap.

Three strategies:

- **Abort:**
Abort transaction, release lock & suspend;
retry later.
- **Commit:**
Finish transaction, release lock & suspend.
- **Side-step:**
Restructure heap to allow concurrent operations.

Commit

- **T/Orbit – “dangerous points”**
 - Interrupt system knows address range of cons “millicode.”
 - Can interrupt anywhere/anytime, unless consing.
 - Cons routine polls for pending interrupts when done.
- **SML/NJ – “safe points”**
 - Interrupt system always defers interrupts.
 - Basic blocks poll for pending interrupts.
 - Interrupt poll uses heap-limit check.

Pros and cons of commit

- **Good**
 - Reduced GC-information requirements. (safe-points)
 - Tolerant of imprecise interrupts.
(safe-points)
- **Bad**
 - Interrupt latency (safe-points)
 - Page fault (both)

**Page fault is
the show-stopper.**

Abort

A clever trick, possibly due to Appel:

```
;; r1 := cons(r2,r3)  
;; fp is frontier pointer
```

```
L1: fp[0] := r2  
     fp[4] := r3  
     r1 := fp  
     fp += 8
```

- Incrementing fp commits the transaction.
- No rollback needed;
to abort, just walk away!
- Resume at L1.

We are exporting OS/thread-global state (fp) into the register set.

How it works

- Interrupt system examines instruction at interrupted pc.
- If consing, scan backwards looking for beginning of sequence.

Locking overhead: 0 instructions!

Pros and cons of abort

- **Good**
 - Can always abort — even during page fault.
 - Good interrupt latency
- **Bad**
 - Terrible interrupt overhead
 - Very painful to implement

Abort II — register locks

Fp is word-aligned \Rightarrow low 2-3 bits available!

Low bit is lock bit.

```
;; r1 := cons(r2,r3)
```

```
;; fp is frontier pointer
```

```
L1: fp += 1
```

```
    fp[3] := r3
```

```
    fp[-1] := r2
```

```
    r1 := fp-1
```

```
    fp += 7
```

Abort II — register locks

Fp is word-aligned \Rightarrow low 2-3 bits available!

Low bit is lock bit.

```
;; r1 := cons(r2,r3)
```

```
;; fp is frontier pointer
```

```
L1: fp += 1          /* Lock heap.          */
    fp[3] := r3      /* Init cdr & trap on GC. */
    fp[-1] := r2     /* Init car.             */
    r1 := fp-1
    fp += 7          /* Unlock & commit      */
```

- Incrementing *fp* *atomically* unlocks & commits.
- Interrupt-system check very cheap.
- Cons overhead: 1 register instruction.

Determine restart address: binary-search on PC vector, add one more instruction to cons sequence, ...

Side-step

Per-thread allocation arenas decouple inter-thread interaction.

Threads must still synchronise with GC.

Memory fragmentation issues arise.

GC breaks locks on continuation-copy,
not interrupt system.

A first implementation

Modified SML/NJ for Sparc: register-locking abort strategy.

Untuned code annotations (4x code size)

Clumsy per-basic-block heap-limit poll.

Write-barrier issues for generational collector:
side-effects allocate!

10% speed penalty.

Unix is barrier to high-quality measurements.

Abort III — A new implementation

Modified SML/NJ for x86.

Can run on raw hardware in ML/OS;
better measurements.

New idea: the “static arena”
combines good points of safe-points & abort.

Performance tradeoff:

- **Safe points**

Locking overhead per-basic-block,
not per-allocation.

- **Abort**

Locking overhead per-allocation,
not per-basic-block.

Atomic fission & the static arena

- *Allocation* atomic;
- *Initialisation* interruptable.
- GC collects partially-initialised blocks
⇒ heavy use of code annotations.
- All allocation in a basic block coalesced;
basic block manages “static” arena.

Splitting the transaction & the SAP

```
fp += 1          /* Alloc 42b for static arena. */
sap := fp-1     /* These insns are atomic;      */
fp += 41        /* rest of b.b. is not.                */
:
sap[0] := r2    /* r1 := cons(r2,r3) */
sap[4] := r3
r1 := sap      /* Commit cons.      */
:
sap[8] := r5    /* r4 := cons(r5,r6) */
sap[12] := r6
r4 := sap+4    /* Commit cons.      */
:
```

3 insn allocation \Rightarrow no pc reset needed!

Alternate allocation sequences

Optimistic-concurrency primitives \Rightarrow
thread can synchronously manage abort;
no pc-reset games required:

L1:

```
r2 := fp      /* Original fp value */
r1 := fp      /* Allocate          */
r3 := fp+8    /* New fp value      */
cswp(r2,fp,r3) /* Attempt fp update */
jnz L1        /* Retry on failure  */
```

Even better: can allocate atomically on x86
with exchange-and-add!

```
/* fp in memory; sap in register */
sap := 8
sap := sap xaddl fp
```

Sparc SAP numbers

Benchmark	Run time	% change
boyer	68.13	-0.8
life	7.62	-3.7
knuth-bend	56.05	-0.5
lexgen	65.08	0.2
mlyacc	47.47	-0.3
vliw	52.38	-5.9
fft	58.16	-11.0
logic	72.69	6.0
simple	14.03	-1.6
mandelbrot	89.85	-6.2
ray	45.87	2.9
barnes-hut	50.50	6.1
ratio-regi	560.56	18.4
count-grap	179.32	1.2
average		0.345

Storage allocation as kernel service

- `sbrk`
- `malloc + sigblock`
- `cons`

FP implementors must commit to implementing own “virtual OS.”

Performance comes from tightly integrating OS & compiler:

- Exporting kernel state to register set;
- pc annotations \Rightarrow GC;
- `cons` as inlined kernel code.

Modern systems must bite bullet and deal with thread schedulers that handle page faults.

Write barriers

We must watch store operations to memory to remember old locations where we store new pointers.

- Exploit type information:
Storing a non-pointer (character, boolean) can be compiled w/o write barrier.
- Store lists:
Now side-effects allocate memory!
- Card marking:
Keep region of mark bits.
- VM page-table tricks:
Super clever! But OS support weak.

Write barriers are the dark side of generational GC.

Real-time / concurrent GC

- Real-time: bound pause times.
- Concurrent: Interleave GC & computation.

We might be willing to pay more cycles if we can spread them out over time.

Clever algorithm due to Ellis, Li & Appel.