MULTICORE PROGRAMMING

Implementing KCAS and reclaiming descriptors

Lecture 12

Trevor Brown

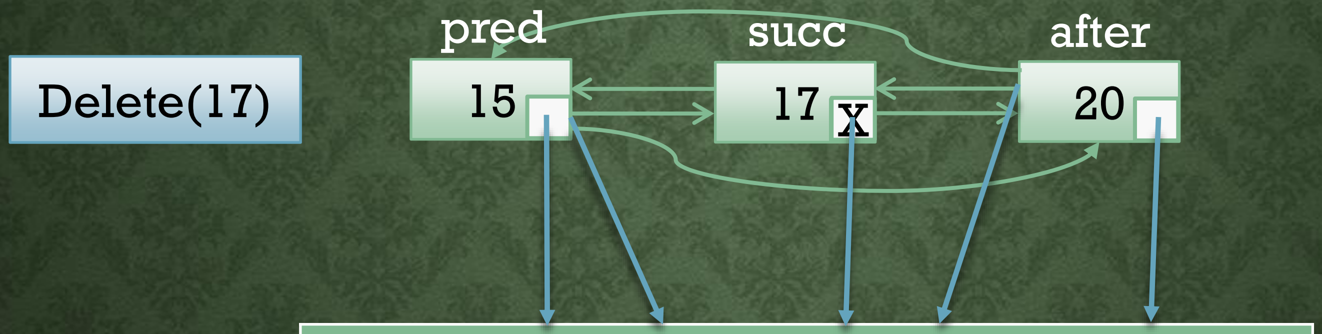
LAST TIME

- Implementing double-compare-single-swap
  - Using **descriptors** and **helping** to guarantee lock-free progress
- Started implementing k-word compare-and-swap

THIS TIME

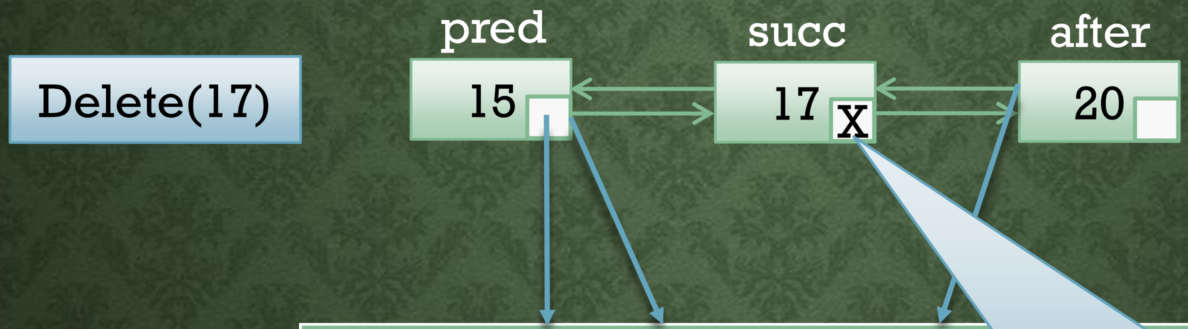
- Finishing the implementation of KCAS
- Reclaiming memory for DCSS and KCAS
  - How to **use** epoch-based memory reclamation
- On the slides (but not in the lecture):
  - Quick intro to some debugging/perf tools

INTUITION: HOW A **SUCCESSFUL** KCAS WORKS: DOUBLY-LINKED LIST AS AN EXAMPLE



KCAS descriptor		
status = Succeeded		
n = 5		
&pred.next	succ	after
&after.prev	succ	pred
&pred.mark	false	false
&succ.mark	false	true
&after.mark	false	false

INTUITION: HOW A **FAILED** KCAS WORKS: DOUBLY-LINKED LIST AS AN EXAMPLE



KCAS descriptor *d*

status = Failed

n = 5

&pred.next	succ	after
&after.prev	succ	pred
&pred.mark	false	false
&succ.mark	false	true
&after.mark	false	false

Failed to change **succ.mark** from **false** to **point to d**!

CAS from **d** back to **expected values**

KEEPING HELPER THREADS IN SYNC

- Key ideas:
- In phase 1 (lock-free "locking"),
  helpers compete to CAS the status from Undecided to Succeeded or Failed
  - Only one helper can "win" and change status
  - Once the status is Succeeded or Failed,
    no more lock-free "locking" should happen
  - I.e., helpers should **no longer** change addresses to point to the KCAS descriptor
  - Accomplish this with DCSS!
- In phase 2 (completion), all helpers **agree** (based on the **status**)
  to change all addresses to new values, or back to their old values ($exp_1 \ldots exp_k$)

USING DCSS IN THE “LOCKING” PHASE

- Threads use **DCSS** to “lock” addresses (storing a pointer to a KCAS descriptor)
  - DCSS addr1 = status field of the KCAS descriptor
  - DCSS exp1 = Undecided
  - DCSS addr2 = address to be “locked” for the KCAS (from KCAS arguments)
  - DCSS exp2 = expected value for that address (from KCAS arguments)
  - DCSS new2 = pointer to the KCAS descriptor
- Semantics of DCSS guarantee:
  - KCAS will successfully “lock” an address <u>**only**</u> if the KCAS status is still **Undecided**
  - Without this guarantee, something called an *ABA problem* can occur. (discussion…)

DISTINGUISHING BETWEEN DESCRIPTORS

- Now that we have DCSS descriptors and KCAS descriptors,
  we must be able to distinguish between them
- Steal **another** bit from each word
  (DCSS uses the least significant bit, KCAS uses 2nd-least significant)
- The two least significant bits tell us whether an address contains
  a value, DCSS descriptor, or KCAS descriptor
    - pack(d): return d | 1 [making it “look like” a DCSS descriptor pointer]
    - packKCAS(d): return d | 2 [making it “look like” a KCAS descriptor pointer]
    - unpack(d): return d & ~3 [zeroing out the bottom two bits]

IMPLEMENTATION

Data structures

```
struct KCAS_desc {
  atomic<word_t>  status;
  word_t          n;
  KCAS_row        row[K];
  char            padding[24];
} __attribute__ ((aligned(64)));
```

```
struct KCAS_row {
  atomic<word_t> * addr;
  word_t           exp;
  word_t           new;
};
```

Code

```
bool KCAS(addr1, ..., exp1,  ..., new1,  ...)
    KCAS_desc * d = new KCAS_desc(addr1, ...);
    d->status = Undecided;
    SortRowsByAddress(d); // to avoid livelock
    return KCASHelp(d);
```

```
word_t KCASRead(atomic<word_t> * addr)
    word_t v;
    do {
      v = DCSSRead(addr);
      if (isKCAS(v)) KCASHelp(unpack(v));
    } while (isKCAS(v));
    return v;
```

Linearize at **last** DCSSRead(addr)

```
bool KCASHelp(KCAS_desc * d)
int newStatus = Succeeded;
if (d->status == Undecided)
| for (int i = 0; i < d->n; i++)
| | word_t val2 = DCSS(&d->status, d->row[i].addr,
| |                    Undecided,  d->row[i].exp,
| |                                packKCAS(d));
| | if (val2 != d->row[i].exp)   // if DCSS failed
| | | if (isKCAS(val2))          // because of a KCAS
| | |   if (unpack(val2) != d)   // a DIFFERENT KCAS
| | |     KCASHelp(unpack(val2));
| | |     --i; continue; // retry "locking" this addr
| | |   // else another helper "locked" for us
| | | else // addr does not contain its exp value
| | |   newStatus = Failed; break;
| CAS(&d->status, Undecided, newStatus);
bool succ = (d->status == Succeeded);
for (int i = 0; i < d->n; i++)
| val = (succ) ? d->row[i].new : d->row[i].exp;
| CAS(d->row[i].addr, packKCAS(d), val);
return succ;
```

Use DCSS to change addresses to point to the KCAS descriptor

Phase 1: lock-free “locking”

Status CAS

Phase 2: completion

```
bool KCASHelp(KCAS_desc * d)
int newStatus = Undecided;
if (d->status == Undecided)
| for (int i = 0; i < d->n; i++)
| | word_t val2 = DCSS(&d->status, d->row[i].addr,
| |                    Undecided,  d->row[i].exp,
| |                                packKCAS(d));
| | if (val2 != d->row[i].exp)   // if DCSS failed
| | | if (isKCAS(val2))          // because of a KCAS
| | |   if (unpack(val2) != d)   // a DIFFERENT KCAS
| | |     KCASHelp(unpack(val2));
| | |     --i; continue; // retry "locking" this addr
| | |   // else another helper "locked" for us
| | | else // addr does not contain its exp value
| | |   newStatus = Failed; break;
| CAS(&d->status, Undecided, newStatus);
bool succ = (d->status == Succeeded);
for (int i = 0; i < d->n; i++)
| val = (succ) ? d->row[i].new : d->row[i].exp;
| CAS(d->row[i].addr, packKCAS(d), val);
return succ;
```

Recall: KCAS just returns KCASHelp(d)

Where should we linearize a successful KCAS?

At the status CAS!

The behaviour of all helper threads, and hence, the outcome of the KCAS, is decided there.

(Crucial points: (1) everything is “locked” at that time, and (2) no thread can see the “old” values after that time.)

Why does this work?

Complicated argument!

Model checking + proof sketch in paper.

Deeper than we need to go.

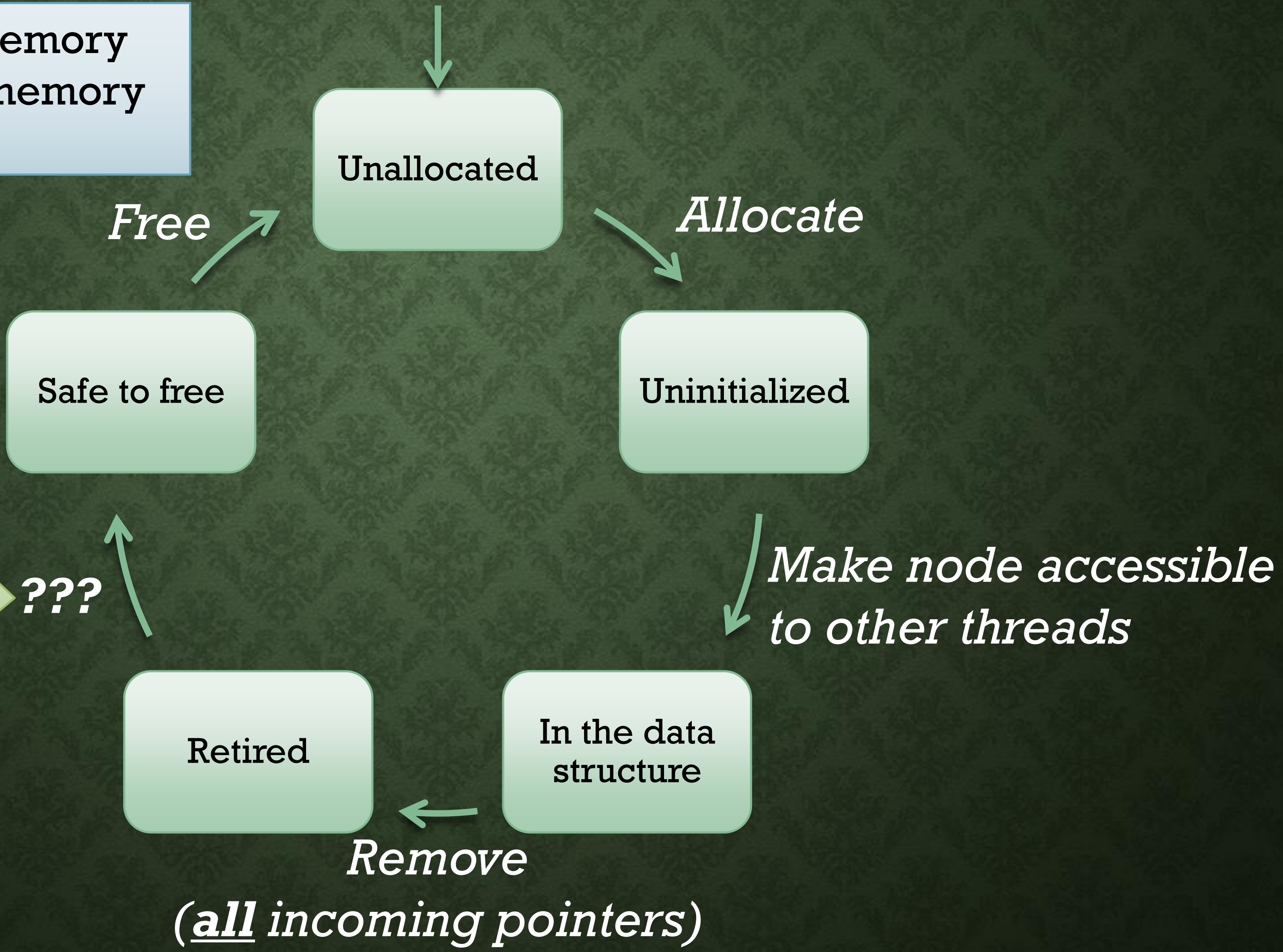
RECLAIMING DESCRIPTORS

LIFECYCLE OF A NODE

Can’t reach from shared memory
+ can’t reach from private memory
= safe to free

Safe memory reclamation problem:

Determine that **no thread** can reach the record by following pointers from private/stack memory



Unallocated

Allocate

Uninitialized

Make node accessible to other threads

In the data structure

Remove
*(<u>**all**</u> incoming pointers)*

Retired

???

Safe to free

Free

EPOCH BASED RECLAMATION (EBR)

- EBR is a relatively **low-overhead** **blocking** solution to the **safe memory reclamation** problem
- Consider a data structure composed of **records** (e.g., descriptors) that should be reclaimed
- Suppose threads **do not** remember pointers to records found in one operation, and then use them in another subsequent operation
  - Instead, when starting a new operation, a thread "**forgets**" all pointers to records, and can **only** access a record by starting at some address in shared memory that is **not** part of a record, and following pointers from there.)
- **EBR interface:**
  - startOp()
    - Must be invoked at the beginning of each operation **before** accessing any **shared** records (i.e., records that have previously been made accessible to other threads)
  - retire(rec)
    - Should be invoked once rec is **no longer reachable** from shared memory
      - **Can** still be reachable from threads' local memories, however… this is fine!
      - Unlike free(), retire will **delay reclamation** until **no thread** has a pointer to node

How is EBR implemented?
Will see later…
Let's see how to **use it**.

USING EBR IN DCSS

DCSS_desc is our "record" type

Do we always access a DCSS_desc by following pointers starting from an address that is **not** part of a DCSS_desc?

Yes! Any DCSS_desc that we access is found by reading an address passed to DCSS/DCSSRead (and this address cannot be part of a DCSS_desc)

```
word_t DCSS(addr1, addr2, e                    )
  DCSS_desc * d = new DCSS_desc(addr1, ...);
  word_t val2;
  while (true) {
    val2 = VAL_CAS(d->addr2, d->exp2, pack(d));
    if (isDCSS(val2)) DCSSHelp(unpack(val2));
    else break;
  }
  if (val2 == d->exp2) {
    DCSSHelp(d); //                        ation
  }
  return val2;
```

startOp() here

retire(d)

else free(d)

```
word_t DCSSRead(atomic<word_t> * addr)
  word_t v;
  while (true) {
    v = *addr;
    if (isDCSS(v)) DCSSHelp(unpack(v));
    else break;
  }
  return v;
```

startOp() here

USING EBR IN **KCAS**

KCAS_desc is our "record" type

Do we always access a KCAS_desc by following pointers starting from an address that is **not** part of a KCAS_desc?

Yes! Any KCAS_desc that we access is found by reading an address passed to KCAS/KCASRead (and this address cannot be part of a KCAS_desc)

No need to worry about DCSS_desc records, as those are completely encapsulated in DCSS (black box)

startOp() here

```
bool KCAS(addr1, ..., exp1, ..., new1, ...)
    KCAS_desc * d = new KCAS_desc(addr1, ...);
    d->status = Undecided;
    SortRowsByAddress(d); // can skip sometimes
    bool ret = KCASHelp(d);
    return ret;
```

retire(d)

```
word_t KCASRead(atomic<word_t> * addr)
    word_t v;
    do {
      v = DCSSRead(addr);
      if (isKCAS(v)) KCASHelp(unpack(v));
    } while (isKCAS(v));
    return v;
```

startOp() here

Note: from the perspective of the KCAS algorithm, the **DCSS object is a black box**.
Reclamation of *DCSS descriptors* is **hidden** in the implementation of DCSS.
(Conceptually, there are **two instances of the EBR algorithm**: one for DCSS, one for KCAS)

TOOLS FOR DEBUGGING AND PERFORMANCE

- Debugging and optimizing concurrent programs is **very** hard. Tools can help!

- Debugging
  - GNU Debugger (GDB)
    - Segfaults, infinite loops
  - Address Sanitizer (ASan)
    - Segfaults, memory leaks
    - 1~2x slowdown
  - Valgrind
    - Segfaults, memory leaks, **memory access errors**
    - many-x slowdown
  - Graphviz
    - **Visualizing** pointer based data structures

- Performance
  - Linux Perftools (perf)
    - Studying cycles, cache misses, instructions, stalled cycles
    - At the whole-application level
  - C/C++ Performance API (PAPI)
    - Precise information from perf, but recorded **within** your program
  - VTune Amplifier
    - Powerful (and now free!) profiler

A lot of errors in concurrent programs manifest as memory access errors! For example, a thread may write a bad value into a pointer because of a concurrency bug, and another thread may then read it.

DEBUGGING TOOLS

USING VALGRIND TO FIND MEMORY ACCESS ERRORS

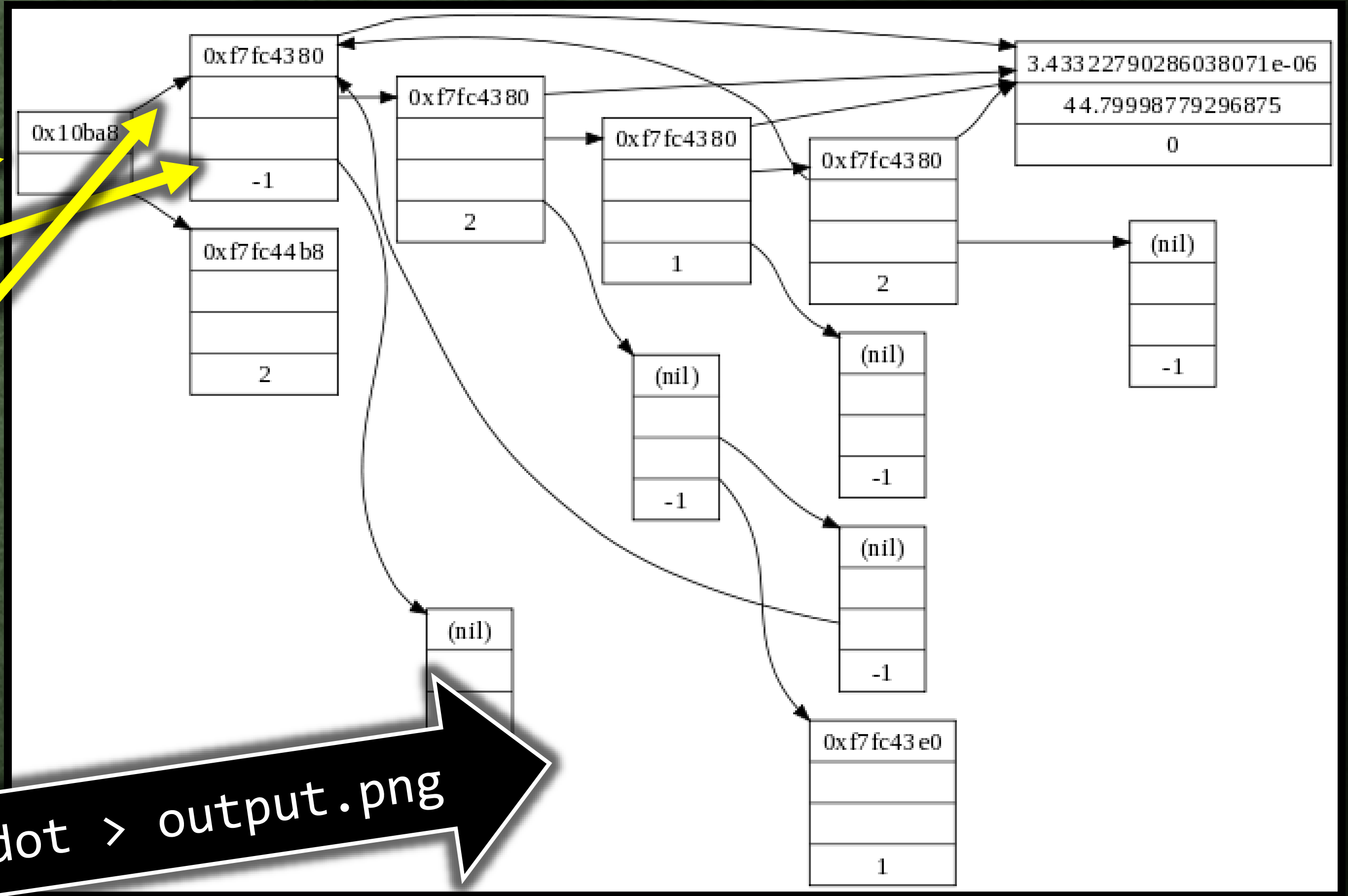
```
$ valgrind --fair-sched=yes ./a1code_segfault/workload_timed.out 4 1000 naive
==107893== Command: ./a1code_segfault/workload_timed.out 4 1000 naive
==107893==
==107893== Use of uninitialised value of size 8
==107893==    at 0x510F0D4: std::thread::join() (in /.../x86_64-linux-gnu/libstdc++.so.6.0.22)
==107893==    by 0x1092DC: void runExperiment<CounterNaive>(...) (workload_timed.cpp:46)
==107893==    by 0x108E3B: main (workload_timed.cpp:70)
==107893==
==107893== Invalid read of size 8
==107893==    at 0x510F0D4: std::thread::join() (in /.../x86_64-linux-gnu/libstdc++.so.6.0.22)
==107893==    by 0x1092DC: void runExperiment<CounterNaive>(...) (workload_timed.cpp:46)
==107893==    by 0x108E3B: main (workload_timed.cpp:70)
==107893==  Address 0x190 is not stack'd, malloc'd or (recently) free’d
...
```

Using Address Sanitizer to check for memory leaks

```
$ g++ -pthread -g -fsanitize=address -static-libasan -fopenmp -O3 ex2_mmult_threads.cpp
$ ./a.out 24
matrix created: 0.02s
randomize call finished: 0.10s
...
multiply call finished: 2.82s

=================================================================
==76549==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 192 byte(s) in 24 object(s) allocated from:
    #0 0x555b294992d8 in operator new(unsigned long) (/home/.../a.out+0xc82d8)
    #1 0x555b294dab9e in matrix::multiply(matrix*, int) (/home/.../ex2_mmult_threads.cpp:12)

SUMMARY: AddressSanitizer: 192 byte(s) leaked in 24 allocation(s).
```

GRAPHVIZ: WHEN YOU JUST NEED TO **SEE** IT

```
digraph g {
  node [
    fontsize = "16"
    shape = “record"
  ];
  edge [];
  "node0" [
    label = "<f0> 0x10ba8| <f1>“
  ];
  "node1" [
    label = "<f0> 0xf7fc4380|<f1>|<f2>|-1"
  ];
  [...]
  "node0":f0 -> "node1":f0 [
    id = 0
  ];
  [...]
}
```



```
$ dot -Tpng input.dot > output.png
```

Tip: try to get your toughest bugs to happen in SMALL data structures, so you can graphviz them

SANITY CHECKING: EXPERIMENT CHECKSUMS

- Important to perform sanity checks wherever you can!
  - Helps to catch obvious (and non-obvious) mistakes
- One good sanity check: checksum based validation
  - Reduce the **data structure** to a number (a **data structure checksum**)
  - Reduce each threads' completed operations to a number (a **thread checksum**)
  - verify that thread checksums **"match"** the data structure checksum
  - (I.e., the work the threads **think** they've done is reflected **in the data structure**!)
- **Creativity needed to come up with good checksum functions**

PERFORMANCE TOOLS

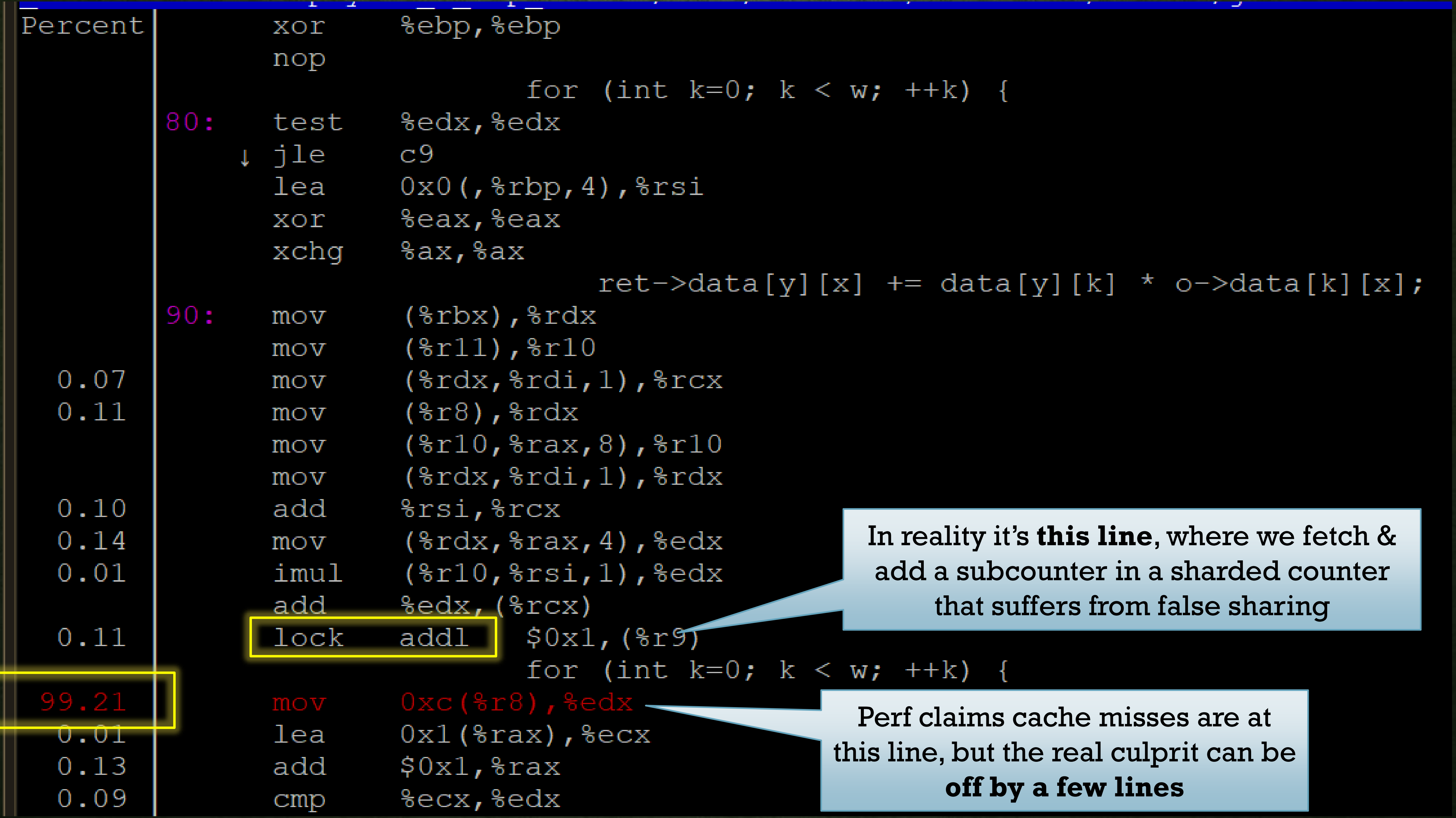
Investigating cache misses with Linux Perftools: perf record

```
$ perf record -e cache-misses ./ex4_counting_events_counter4.out 24
matrix created: 0.00s
randomize call finished: 0.03s
...
number of additions = 1000000000
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.802 MB perf.data (20591 samples) ]

$ perf report
```

Investigating performance with Linux Perftools: perf *report*

```
Samples: 20K of event 'cache-misses', Event count (approx.): 543388
Overhead  Command          Shared Object           Symbol
  22.51%  ex4_counting_ev  ex4_counting_[…].out   [.] _ZN6matrix8multiplyE...
   5.64%  ex4_counting_ev  [kernel.kallsyms]       [k] decay_load
   5.07%  ex4_counting_ev  [kernel.kallsyms]       [k] native_sched_clock
   5.02%  ex4_counting_ev  [kernel.kallsyms]       [k] __update_load_avg_se...
   4.21%  ex4_counting_ev  [kernel.kallsyms]       [k] perf_event_alloc.par...
   3.75%  ex4_counting_ev  [kernel.kallsyms        [k] _raw_spin_lock
   3.73%  ex4_counting_ev  [kernel.kallsyms        [k] cgroup_rstat_updated
   3.34%  ex4_counting_ev  [kernel.kallsym         [k] task_tick_fair
          [...]
```

Interactive console… select this line and press [ENTER] twice…



```
Percent │       xor    %ebp,%ebp
        │       nop
        │                   for (int k=0; k < w; ++k) {
        │ 80:   test   %edx,%edx
        │     ↓ jle    c9
        │       lea    0x0(,%rbp,4),%rsi
        │       xor    %eax,%eax
        │       xchg   %ax,%ax
        │                        ret->data[y][x] += data[y][k] * o->data[k][x];
        │ 90:   mov    (%rbx),%rdx
        │       mov    (%r11),%r10
   0.07 │       mov    (%rdx,%rdi,1),%rcx
   0.11 │       mov    (%r8),%rdx
        │       mov    (%r10,%rax,8),%r10
        │       mov    (%rdx,%rdi,1),%rdx
   0.10 │       add    %rsi,%rcx
   0.14 │       mov    (%rdx,%rax,4),%edx
   0.01 │       imul   (%r10,%rsi,1),%edx
        │       add    %edx,(%rcx)
   0.11 │       lock   addl   $0x1,(%r9)
        │                   for (int k=0; k < w; ++k) {
  99.21 │       mov    0xc(%r8),%edx
   0.01 │       lea    0x1(%rax),%ecx
   0.13 │       add    $0x1,%rax
   0.09 │       cmp    %ecx,%edx
```

In reality it's **this line**, where we fetch & add a subcounter in a sharded counter that suffers from false sharing

Perf claims cache misses are at this line, but the real culprit can be **off by a few lines**

C/C++ LIBRARY: PAPI

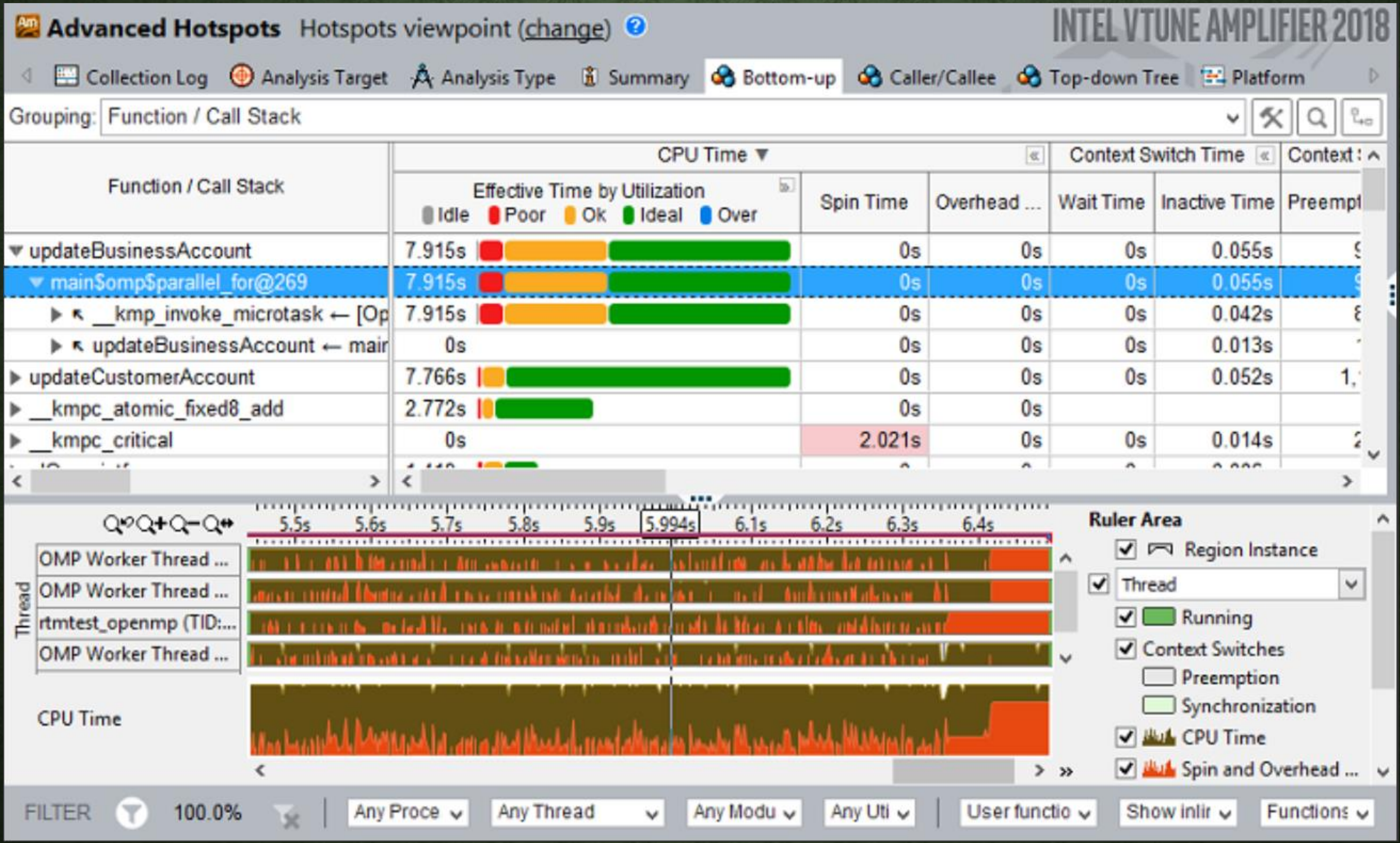
- https://icl.utk.edu/papi/
- Gives access to most of the same stuff as **perf stat/record/report,** *but programmatically inside your own code*
- Can **always** include some measurements in your runs
  - Fast --- no real overhead on Intel for ~2-4 performance monitoring counters
  - Can easily measure *only part* of your execution (skip measuring setup/teardown)
  - Can present results in a nice format (e.g., L3 cache misses PER data structure operation)

```
total throughput                 : 94906203
PAPI_L1_DCM=36.4196
PAPI_L2_TCM=24.8331
PAPI_L3_TCM=11.8196
PAPI_TOT_CYC=5515.5
```

These are all “per data structure operation”

WHEN YOU NEED A REAL PROFILER: VTUNE

- Surprisingly, Intel's VTune is free. Even for commercial use!
- Great profiler for seeing what threads are doing throughout your execution
- Surprisingly easy to learn and use, even in complex scenarios…
  - Profiling code that runs locally is trivial
  - Profiling code that runs remotely:
    - ~6 hours invested to learn the idiosyncrasies of VTune + remote execution



Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼ Effective Time by Utilization Idle Poor Ok Ideal Over	Spin Time	Overhead ...	Context Switch Time Wait Time	Inactive Time	Context : Preempt
▼ updateBusinessAccount	7.915s	0s	0s	0s	0.055s	9
▼ mainompparallel_for@269	7.915s	0s	0s	0s	0.055s	9
▶ ↖ __kmp_invoke_microtask ← [Op	7.915s	0s	0s	0s	0.042s	8
▶ ↖ updateBusinessAccount ← mair	0s	0s	0s	0s	0.013s	1
▶ updateCustomerAccount	7.766s	0s	0s	0s	0.052s	1,1
▶ __kmpc_atomic_fixed8_add	2.772s	0s	0s			
▶ __kmpc_critical	0s	2.021s	0s	0s	0.014s	2