Lecture 11

Checking Concurrent Prgms

Zvonimir Rakamarić
University of Utah

Announcements

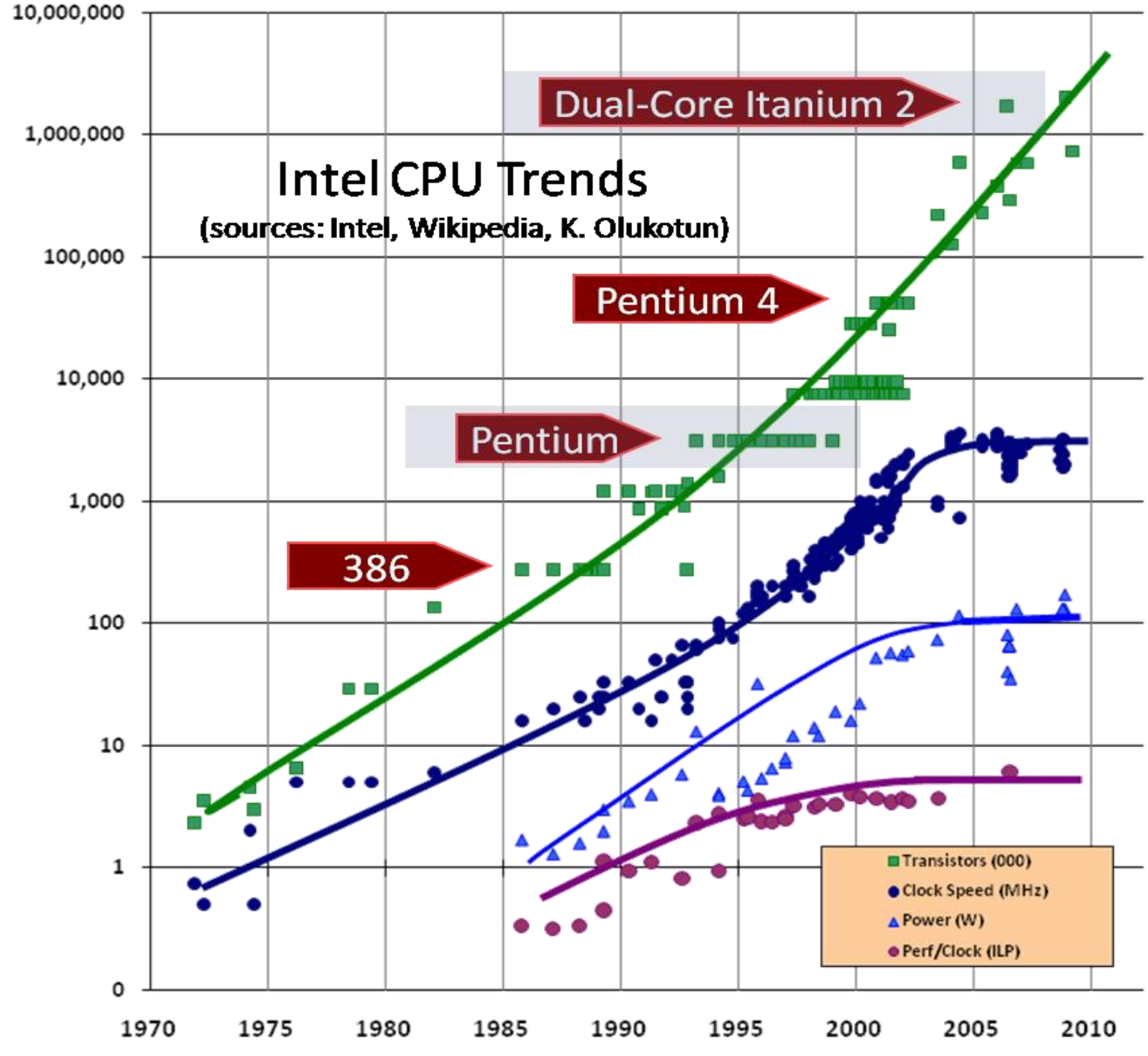
- Student course feedback
- Last homework is due Thu evening
  - Submit your solution to github
- SAT competition
  - Deadline is noon tomorrow (Tue)
  - We will use your last commit or message me commit hash
- Visited Univ. of Utah data center – cool!
- Last class on Wed
  - Announcing results of the competition
  - Pizza party

A long time ago (in a galaxy far, far away)...

- Doubling of computing performance every two years
  - Moore's Law
    - 2x transistors every 2 years
  - Dennard scaling
    - Transistors will be faster and lower energy

- Computer scientists and practitioners alike took for granted that their scaling problems would be solved if they simply waited a year or two

Computing Revolution



Dennard + Moore
= 2x performance
every 2 years

Chart originally from The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, 2009

Why Multicore in PCs?

- We were forced to multicore when Dennard scaling broke down around a decade ago
  - Could not increase frequency any more, but we could still put more transistors on chips

- Moore's law
  - The number of transistors in chips doubles every two years
  - Seems to be failing too [http://www.networkworld.com/article/2949034/]
    - Entering *dark silicon* era
    - Approximate heterogeneous computing

Today Concurrency is Pervasive

- Old problem of computer science
  - Since ancient supercomputers
- Today
  - Multi-cores even in cell phones
  - Many-cores in desktops
- Most programs are concurrent
  - At least the ones you care about

Many Paradigms, Languages, Libraries

- MPI
- Pthreads
- OpenMP, CUDA, OpenCL
- Cilk
- Chapel
- Erlang
- Scala, Java, C#
- Go

```c
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

Concurrency is Hard I

- Inefficient (dumb) concurrency is easy
  - Big fat lock around everything
  - Poor performance
- Efficient concurrency is hard
- A concurrent program should
  - Function correctly
  - Maximize throughput
    - Finish as many tasks as possible
  - Minimize latency
    - Respond to requests as soon as possible
  - While handling nondeterminism in the environment

Concurrency is Hard II

- Huge number of possible thread interleavings (or schedules)
- How many interleavings are possible in a concurrent program with *n* threads where each thread has *k* instructions?

$$(n*k)! \,/\, (k!)^n \geq (n!)^k$$

- Exponential in both n and k!
- Example: 5 threads with 5 instruction each

$$25! \,/\, 5!^5 = 6.2336074e{+}14$$

$$= 623 \text{ trillion interleavings}$$

Concurrency is Hard III

- Concurrent executions (thread interleavings) are highly nondeterminisitic
- Stress testing
  - Trying to explore many different thread interleavings by creating hundreds of threads
- Stress testing is highly inefficient
  - Some concurrency bugs occur only in certain thread interleavings
    - Finding the “right” thread interleaving is pure luck
    - No notion of coverage
    - Running for days, even months

Concurrency Bugs

- Rare thread interleavings result in Heisenbugs
  - Difficult to find, reproduce, and debug
- Observing the bug can “fix” it
  - E.g., likelihood of interleavings changes when you add `printf` statements
- A huge productivity problem
  - Developers and testers can spend weeks chasing a single Heisenbug

Common Concurrency-Related Issues

- Data races
- Atomicity violations
- Assertion violations due to thread interleavings
- Wrong result or crash
- Non-determinism

Data Race

- Data race is a simultaneous (concurrent) access to the same memory location by multiple threads, where at least one of the accesses modifies the memory location

- Important class of bugs and the main focus on this lecture

```
int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
  assert(Global == 0);
}
```

Does this assertion always hold?

Data Race

- How would you test your concurrent program for data races?

Short Break: AptLab Tsan VM

- Go to https://www.aptlab.net/

- Click on “Change Profile” and select “tsan_vm”

```
int Global;

void *Thread1(void *x) {
  int y1 = Global;
  y1++;
  Global = y1;
  return NULL;
}

void *Thread2(void *x) {
  int y2 = Global;
  y2--;
  Global = y2;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
  assert(Global == 0);
}
```

Does this assertion always hold?

```
int Global;

void *Thread1(void *x) {
  Global = 1;
  return NULL;
}

void *Thread2(void *x) {
  Global = 1;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
  assert(Global == 1);
}
```

Does this assertion always hold?

Benign Data Race

- There is no such thing as a *benign* data race
- C standard: program behavior is *undefined* if there is a data race
  - Compiler can do anything!

- If there are no data races, all sequential optimizations are safe
- If there is a data race, all bets are off

- See: H. J. Boehm, “How to miscompile programs with "benign“ data races,”, USENIX Conference on Hot Topic in Parallelism, 2011

Fixing Data Races

- Introduce *synchronization* between threads to ensure *mutual exclusion* between critical parts of your code
  - Critical parts are often called *critical section*
  - Only one thread at a time can be executing a critical section
- Common form of synchronization are locks
  - There are many others

```
int Global;
pthread_mutex_t mtx;

void *Thread1(void *x) {
  pthread_mutex_lock(&mtx);
  Global++;
  pthread_mutex_unlock(&mtx);
  return NULL;
}
void *Thread2(void *x) {
  pthread_mutex_lock(&mtx);
  Global--;
  pthread_mutex_unlock(&mtx);
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_mutex_init(&mtx, 0);
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL); pthread_join(t[1], NULL);
  assert(Global == 0);
  pthread_mutex_destroy(&mtx);
}
```

Does this assertion always hold?

Thread Sanitizer (TSan)

- TSan is a tool for finding data races in C/C++ (and Go) programs
- Regularly used by Google on large code bases (think Chrome browser)

Overview of TSan Algorithm

- It instruments your program during compilation
  - All memory accesses
  - Synchronization
- You have to run your instrumented binary
- Instrumentation tracks memory accesses and synchronization operations (e.g., mutexes)
- If there is no synchronization between two accesses, it reports a data race

Exercise

- Try out TSan in AptLab

- See README in /local/tsan-exercises

Conclusions

- Data races can be really bad and are hard to discover
- Data race checkers are highly advantageous for program understanding
- Crucial tool when something appears to be wrong
  - Nondeterministic crash
  - Cannot replay results or reproduce a crash or assertion violation
- Apply data race checker even when everything seems ok
  - Maybe your tests are missing a race

Useful Links

https://computing.llnl.gov/tutorials/parallel_comp/

https://computing.llnl.gov/tutorials/pthreads/

http://clang.llvm.org/docs/ThreadSanitizer.html