Analysis of Inherent Randomness of the Linux kernel

Nicholas Mc Guire (DSLab Lanzhou University, China)
Peter Okech (Strathmore University, Kenya)
Georg Schiesser (Opentech, Austria)

11th Real-Time Linux Workshop, Dresden – Sept 2009

Agenda

- Introduction
- Concept of Inherent randomness
- De-randomization
- Timestamp precision
- Conclusion and future work

How do we guarantee predictability?

- Tackling the problem of real-time postulates deterministic hardware and software
- The approach to guarantee predictability is to hunt for latency maximum and eliminate it
  - Typically identified with a particular code path or event sequence

Questions to Think About!

- How deterministic is the execution of code on modern super scalar CPUs ?
- How predictable is the overall system if a complex OS (such as GNU/Linux) is run on top of modern CPUs?

Inherent Randomness

- We claim that there is a certain level of randomness that is associated with complexity
- Some of the jitters in code execution time can be attributed to this inherent non-determinism
  - And not specific code path

Sources of Indeterminism

- The source of indeterminism of interest to us are those that we classify as either **internal** and **intentional**
- Internal indeterminism arise from the direct or indirect referencing of global variables by an application
  - Global variables include free shadow registers, TLB, BTB, available cache and memory, timeout in communications e.t.c

Sources of Indeterminism (2)

- Cases when an application uses random data for its decisions is can be regarded as intensional non-determinism, and this may include:
  - Use of random numbers, asynchronous event timestamp, error conditions
- These sources are amplified by concurrency and asynchronous events

Impact of Indeterminism

- These (and other sources of) non-determinism means that individual application code, while exhibiting a well defined local state, has no deterministic global state.
  - We are unable to predict the actual behavior of the application

- We will illustrate the indeterminism using two examples
  - Timing of instructions
  - printf() function

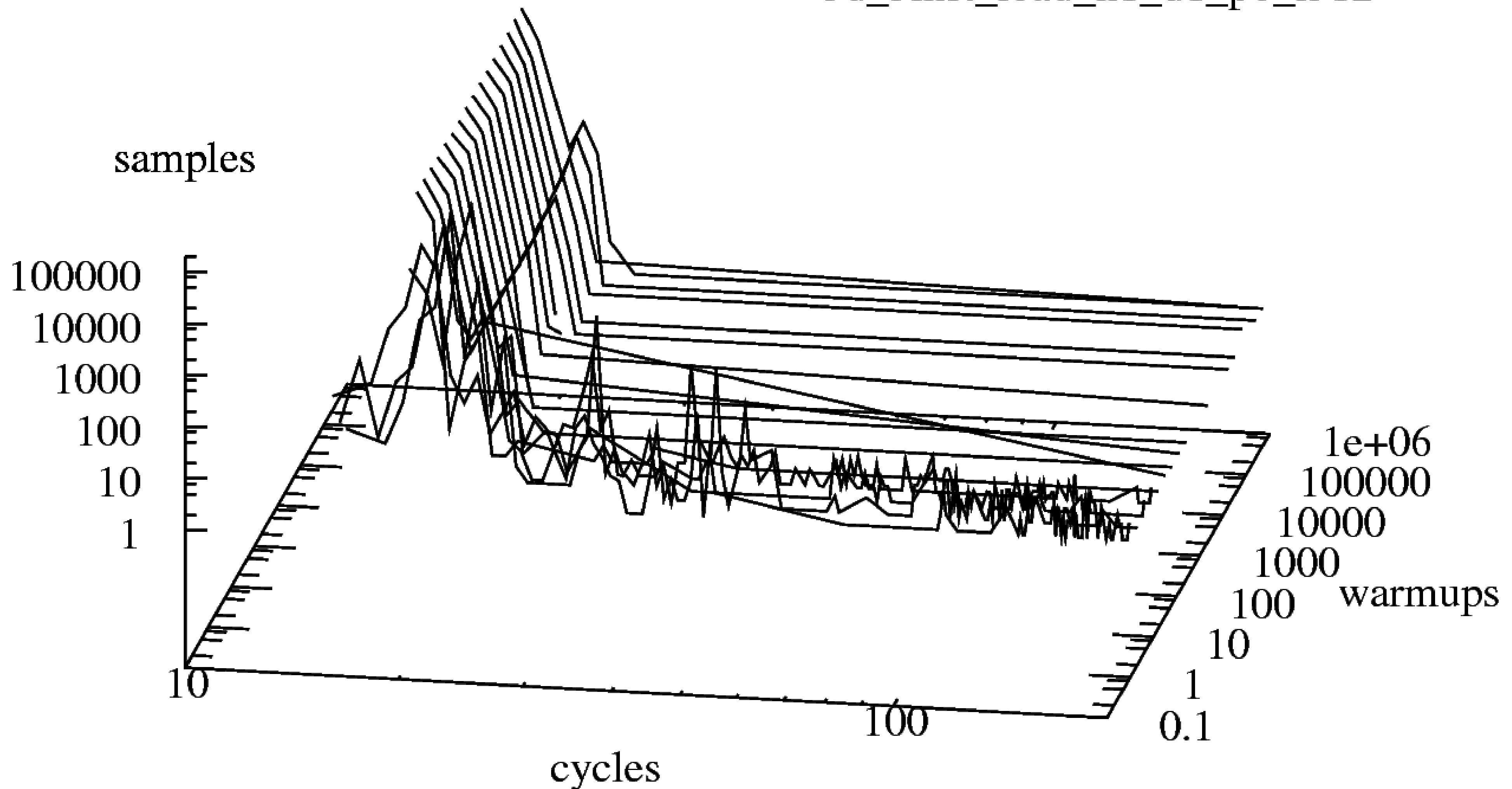
Example 1 - Timing

- The code consists of 5 integer instructions
  - First in a warmup loop to ensure they are cache hot
  - Then measure the times of the final execution
- Results show that the code never reach a constant execution time

Code – 5 integer instruction Timing

```
__asm__ __volatile__("cli":::"memory");
for (j = 0; j < w; j++){
    x1 = 1;
    x2 = x1 * 1;
    x3 = x1 * 1;
    x3--;
    dummy += x3 / 4;
}
__asm__ __volatile__("cpuid\n\t" \
                     "rdtsc\n\t":\
                     "=A" (start));
x1 = 1;
x2 = x1 * 1;
x3 = x1 * 1;
x3--;
dummy += x3 / 4;

__asm__ __volatile__("cpuid\n\t":\
                     "=A" (stop));
__asm__ __volatile__("cli":::"memory");
timestamps[n++] = ((long) (stop - start));
```

Example 1 – Timing Results



Example 2 – printf()

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MSG "Hello World\n"

int main(int argc, char **arg){
   int ret;

   ret = printf(MSG);
   if (ret == strlen(MSG))  {
      return EXIT_SUCCESS;
   } else {
      return EXIT_FAILURE;
   }
}
```

Example 2 – printf()

- The code exhibits non-determinism
  - At application level, printf() could fail
  - At system level, spawning the application could fail
  -
- There are hundreds of points in the code where failures are possible
  - A guarantee of output can only a certain probability
  -
- Main source of non-determinism is not the scope of application code!

Main Goal

- We seek to establish a real-time metric by demonstrating coupling between instruction/CPU complexity and randomness
- We introduce two metrics
  - Inherent system randomness
  - Timestamp precision
- These metrics could be considered as lower bounds for any high level metrics (such as Interrupt latency, WCET)

Inherent Randomness

- To demonstrate the inherent randomness in complex hardware we
  - developed a software RNG based on hardware non-determinism
  - Performed formal analysis of random bit-stream produced
  - Compared the results over a spectrum of HW of varying complexity (currently only on IA)

TRNG Code Outline

```
allocate_buffer()
mlockall()
touch_buffer()

while(n < stream_size){
   buffer[n++]=get_bit_from_tsc(bit)
}

write_buffer_to_file()
```

Code for the TRNG

```
__inline__ unsigned long long int hwtime(int shift)
{
   unsigned long long int x,res;
   int i;
   int bit=1;
   res=0;

   bit<<=shift;
   for(i=0;i<32;i++){
      __asm__ __volatile__("rdtsc\n\t":"=A" (x));
      res|=(((x&bit)>>shift)<<i);
      usleep(delay);
   }
   return res;
}
```

TRNG results

- If underlying system is deterministic, the code will yield a non random sequence
- The code surprisingly produce good random sequence of bits
  - If run in a tight loop, the sequence is random though show a pattern
  - The quality of randomness reaches very high quality if call to usleep() allows execution of unknown code
    - OS randomizing access patterns

TRNG results (2)

- Based on the test from random.org, it should be noted that
  - Chi square clearly and reliably in the random range
  - Entropy in the range of hardware solutions
  - arithmetic mean in a reasonable range
  - Monte Carlo estimation of Pi
  - serial correlation negligible

Typical Run Results

```
trng.c 128 bytes (Core Duo 2)
 Entropy = 7.996973/byte.
 compres = 0 %.
 chi sqr = 19.63.
 Arit mean = 128.2059
 Mont Car Pi = 3.137520601, err. 0.13 %
 Ser. Corel. = 0.003759
```

- Test from the TestU01 (L'Ecuyer, 2002) test suite also confirmed that quality of randomness is high

Software TRNG comparison

- The bit sequence produced by the software TRNG was compared with
  - Geiger-Muller tube detector of background radiation (hotbits.org)
  - Thermal noise probe (random.org)
  - /dev/random (Core Duo 2 2.6.26 Debian)
  - /dev/urandom (Core Duo 2 2.6.26 Debian)

De-randomization

- We needed to confirm that the inherent randomness can be attributed to the CPU
  - This required that code used does not exhibit random execution time

Constraints

- Constraints needed to achieve predictable execution time could include
  - Warm up loops to ensure cache hot code and data
  - Simple set of instructions
  - Use of local data that fit into a single L1 cache line
  - Interrupt disabling
  - Well selected CPU frequency for constant execution time
  - No SMP
  - Serialized instruction
  - Tuned loop to fit instruction pipeline

De-randomizing code

```
warmup_loop {
   sequence
}
rdtsc
sequence
rdtsc
```

Comments on De-randomization

- Even with these efforts, it is not possible to achieve constant execution times
- What we see is the inherent variance of the CPU execution time for a given simple sequence of instructions
- De-randomization is not practically feasible!
  - Any real life code should exhibit inherent randomness

Timestamp Precision

- A fundamental requirement for time based decision making is that an event is precisely timestamped.
  - In an RTOS no decision can be more precise than the timestamp capability
- Timestamp precision depends on
  - Time source resolution
  - Inherent randomness of hardware and software
  - Isolation of time-sampling code

Measuring Timestamp Precision

- To measure the timestamp precision
  - we run two consecutive calls to “rdtsc”
  - Calculate the deference
  - Search for max/min values
- For a given setting (priorities and scheduling policies), one can get an overview of lower bounds of timing at code level
  -
- Also could give a suitable lower bound for OS level scheduling jitters

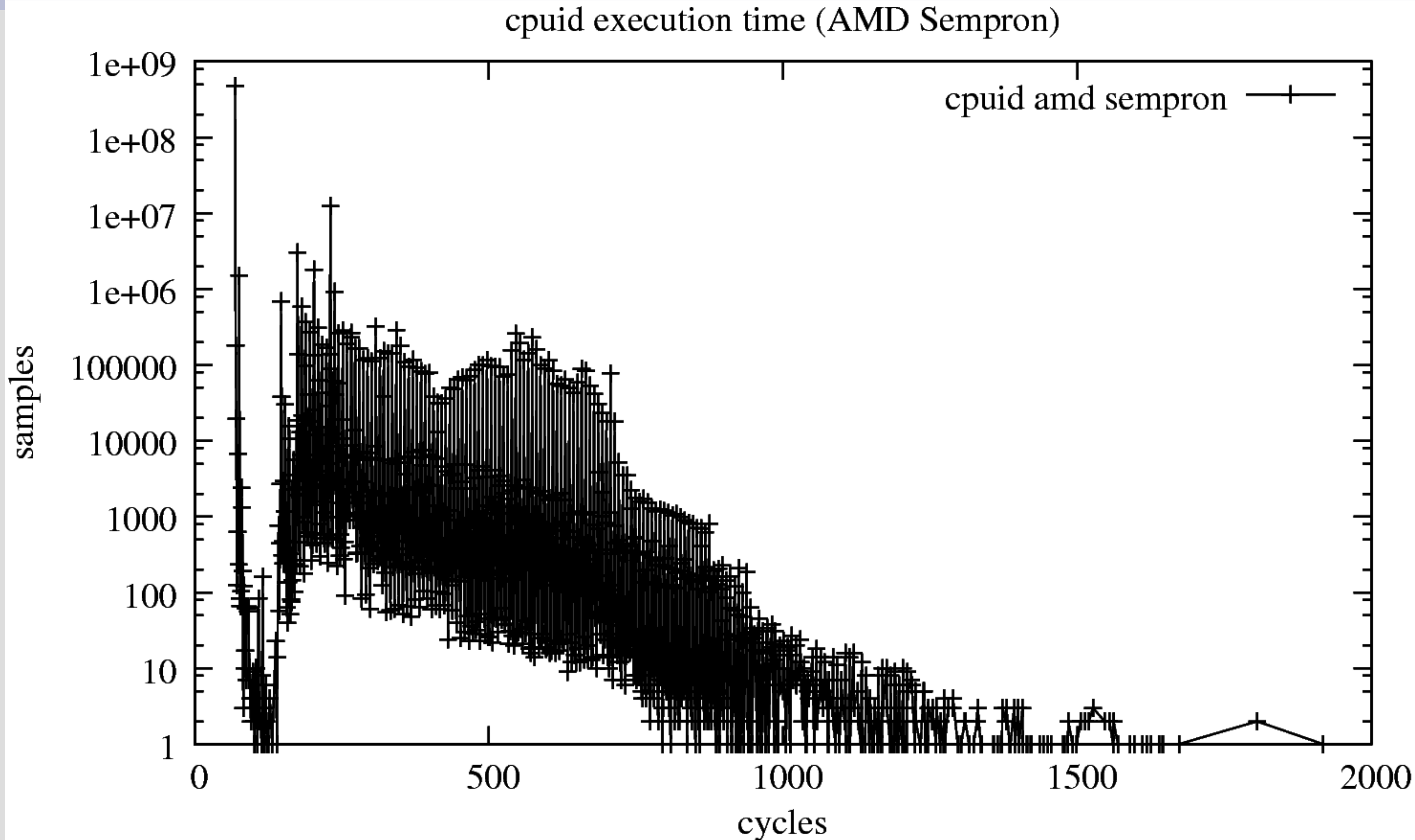
Timestamp Precision code

```
while (n < loops)
{
    unsigned long long index=0;
    usleep(1);
    hwtime2 = rdtsc();
    hwtime1 = rdtsc();

    jitter = hwtime1 - hwtime2;
    index = jitter /scale;

    if (index > GRAPH_SIZE) {
        out_of_bounds = 1;
    } else {
        graph[policy][index] +=1;
    }
    n++;
}
```

Reading the TSC

- We have not used the a serializing instruction (cpuid +rdtsc or rdtscp)
- Serializing instructions using cpuid have a profound negative impact on timestamp precision
  - Is a main limitation
  - cpuid can take up hundreds of cpu-cycles

Impact of cpuid to read the TSC



Conclusion

- Modern CPUs are inherently random
- Complex general purpose OS amplifies this inherent randomness substantially
- A set of acceptable metrics to describe these basic properties (inherent randomness and time precision) is required.

Conclusion (2)

- Practical conclusions
  - Real time metrics must take into account the inherent randomness of modern computing systems
  - A statistical approach to performance measurement is the only meaningful way

Conclusion (3)

- The by-product of this work, the software TRNG can be used
  - For the initialization of the random number pool at boot time or
  - To generate entropy for systems that do not have sufficient sources from asynchronous events

Future work

- Cover inherent randomness in more depth
- Investigate different models that allows for good estimations of execution times for real-time systems based on complex hardware/software

Thank you

Questions?