Analysis of inherent randomness of the Linux kernel

Nicholas Mc Guire, Peter Okech, Georg Schiesser
Distributed & Embedded System Lab, SISE, Lanzhou University, China
Tianshui South Road 222,Lanzhou,P.R.China
mcguire@lzu.edu.cn,POkech@strathmore.edu,georg@opentech.at

Abstract

While analyzing latency data from real-time Linux variants we found that there are distinct parts to the system jitter - those that can be attributed to software constructs and those that are inherent in complex software systems running on non-deterministic hardware. Initial investigations focussed on explaining these results by considering various caches (L1,L2,BTB,TLB,etc) [12]. While this seemed to allow some level of explanation it did not satisfactory explain the distributions found during analysis.

Essentially hunting for the maxima of latency, which was the common initial approach, only can detect grave latency issues, like excessively long holding of locks - but it is not able to detect low-level latency causes like miss-alignments, or short term locks that are in a hot path and thus contribute significantly to the systems overall latency and jitter. Further the maxima - if the assumption of inherent randomness hold - are not associated with a specific code-path but rather with the code-path being executed in a specific, system level, context - thus we believe that a statistic approach to tracing latency is needed.

Basic analysis of real time behavior in complex software systems can be split roughly into the following parts:

- Timestamp precision - how precisely can an event be associated with a timestamp from a specific clock-source
- inherent randomness - how non-predictable is the execution of functionally deterministic code.

The first can be quite nicely measured (or rather estimated based on measurements) - the second is a bit more complicated as there currently is not even a well accepted definition, nor a practically meaningful metric.

These two factors, we believe, can form a useful constraint for the lower-bounds of timing behavior that can be achieved in complex software systems - be that scheduling jitter, interrupt latency, or bandwidth variance.

In this paper we will present the current state of our assessment along with an argument why we believe that inherent randomness is present and of what quality this randomness actually is based on preliminary evaluation of a random number generated (RNG) derived from our timestamp measurement code.

KeyWords: inherent randomness, non-determinism, randomness, complex systems

1 Introduction

Real Time Operating Systems (RTOS) are typically identified with determinism and predictability. The prime way to tackle the problem of RT seems to be to postulate a deterministic hardware system and put on top of it a deterministic software blob - albeit neglecting the limitations of the abstractions behind both the hardware and software. Based on this one then starts hunting for the cause of excessive latency or jitter in a RTOS and tries to identify it with particular code-paths or event sequences. While this is to a certain extent successful - notably as long as there are specific code-paths that need improvement, there is a certain point at which this "hunting-for-maxima" seems to yield no significant improvement. In this paper we will outline our findings with respect to inherent randomness of complex hardware/software systems and argue that there is a certain level of randomness that is associated with the complexity rather than the specific code path and for complex hardware/software systems - GNU/Linux on mainstream super-scalar COTS CPUs - we need to develop metrics and methods to live with this inherent randomness rather than try to fight it.

1.1 Related Work

J. Viega in [3] focussed on security related random number needs, and outlines what methods seem suitable to provide reliable random numbers, though these are based on a seed which is from a true random source (i.e. OS entropy pool fed by interrupt related entropy [4]). The prime conclusion from this paper, from the perspective of our work, is that there are sound algorithms available, notably universal hash functions, that would allow to attain good randomness in a security sense in general purpose OS. A further note in this paper that our contribution also suffers from is that there "..there is not a consistent set of requirements or terminology between different solutions".

Sameer Niphadkar and Matt Davis in [2] describe there approach of using concurrent threads and the indeterminism in the scheduling of a complex OS to produce random numbers through thread synchronization timing variance. They describe this approach being neither a TRNG nor a PRNG - "Threads on the contrary lie somewhere in the middle between the true and pseudo random number generators." In their publication [1] they brand this approach as a "Pseudo Random Number Generator" while we believe there is sound evidence for such an approach being a True Random Number Generator albeit with some deficits that may need fixing in the specific implementation.

With respect to metrics, the most relevant work we have been building on is the work from Pierre L'Ecuyer [6] which provides a extensive set of statistical test utilities to evaluate the output of random number generators. We believe that a subset of these tests along with a well specified procedure could constitute a good basis for a generally acceptable metric for system level inherent randomness.

2 Sources of non-determinism

Any system has global variables in the one or other form, in complex systems there are many such global variables (think of free shadow registers, TLBs, available cache lines, memory ... timeouts in communication, etc.) many of these global variables are locally references directly or indirectly.

- direct reference - a send operation on a queue that can block
- indirect reference - a system call that internally allocates memory dynamically and thus can take largely diverging amounts of time depending on the systems state
- an instruction that needs to evict a L1 I line before being performed
- a my-op that gets stalled due to a (functionally) unrelated pipeline condition
- a branch in the my-ops that is once taken and once not due to the BTB being exhausted by functionally and temporally unrelated code execution paths.

Note that we are not referring to any external or asynchronous events yet in this list - so we refer to this as internal sources of non-determinism.

This for it self still is not non-deterministic from the perspective of the individual application. At the moment where we have a concurrent preemptible system the picture dramatically changes. The local state (application) actually can be revisited, but its dependency on global state makes the outcome indeterminable. For the temporal domain this is a well accepted fact - after all race conditions in software are a much feared fault for decades now and anybody hunting down such a bug knows how undetermined the temporal precision of full featured preemptive operating systems are. Notably the introduction of the lock-dependency validator in mainline 2.6 had revealed a number of race conditions that had not yet been discovered at runtime by anybody, even though they had been present for a substantial time in the kernels code-base.

2.1 Intentional non-determinism

In some cases we know that data used for decisions is random - some may be:

- random numbers (i.e. from a TRNG)
- asynchronous event timestamps
- global conditions (i.e. if(in_interrupt()) derived from asynchronous events
- error conditions (i.e. checking return values) - or is anybody expecting a deterministic failure rate of printf ?

These and other sources of non-determinism exist at even simply applications levels - paired with concurrency this de-facto means that individual application code - while exhibiting a well defined local state - has no deterministic global state and we are unable to predict the actual behavior of even simple applications.

Example: timing of a 5 integer instruction (C-level)

This code run with interrupts disabled, executes 5 integer instructions in a warmup loop - this ensures they are cache hot - and then times the final execution - the plot over the warmup loops indicate that it takes quite a few warmup loops to get the system into a more or less well-defined state - never the less it never reaches a constant execution time.

```
__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__("rdtsc\n\t":\
                     "=A" (stop));

__asm__ __volatile__("sti":::"memory");

timestamps[n++]=((long)(stop-start));
```

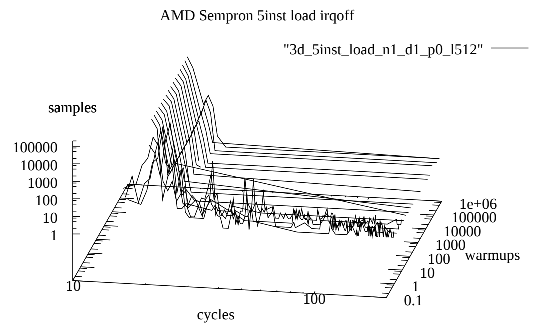

FIGURE 1: *5 instruction execution time with disabled interrupts (AMD Sempron)*

In a modern operating system even a trivial hello world:

```
#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;
        }
}
```

exhibits non-deterministic properties. At the application level printf could fail, at the system level the entire process of spawning the application could fail and this program might actually never be executed at all. If one looks at the possible points of failure of this trivial application on a modern UNIX operating system there are actually a few hundred points in the code where failures are possible (coded fault handling) each of which has only a very small probability of actually ever being seen - never the less we only can guarantee this hello wold to actually output the intended "Hello World" string with a certain probability and even if we were to analyze the entire OS level code (kernel, libc, shell, etc) involved we would not be able to guarantee its execution. The main sources of non-determinism of this trivial application are not in the scope of the application code - in fact one rarely sees anybody bothering to catch the return value of a printf as it simply is assumed to behave well - so most readers will feel this hello world is a bit artificial code. What this does exhibit though is that the local state transition of this trivial example (with its terminal state being success or failure) is dependant entirely on the OS level component and we are unable to actually predict there behavior other than statistically.

2.2 The issues with defining real-time

We are not going to go into the multitude of real-time definitions available, there are many and new ones being added (i.e. cooperate real-time recently hit the stage) - the only contribution to this discussion that we would like to add is that the question of determinism (hard real-time) vs probabilistic definitions (soft real-time,firm real-time) is maybe simply missing the point as they are concerned with two different system descriptions. The hard real-time definitions are focussed on functional determinism - neglecting problems of system failures (i.e. random hardware faults, or meteorites striking the test-system), while the soft real-time definitions are focusing on observations in systems where not all components are known (the question if they could be known or not is not so essential).

The claim we are making here is that at system level (a computing system in a real-world setup) there are always potential failure scenarios that invalidate any deterministic hard real-time definition - although one may be able to reach very high levels of confidence (lets say less than 10E-9 failures per hour). So we are considering real-time a probabilistic quality of a system and are going to present recent tests to argue why this approach makes sense to us. I would like to emphasis that it is not relevant if the system might be deterministic, the question is only if we actually are able to formulate this property and the initial state of the system in a way that allows us to predict the behavior - which would amount to a deterministic hard real-time system - or if the inability to define the initial state and/or formulate the systems behavior prohibit us from determining the "next state" of the system with absolute certenty. This inability need not be based on physical constraints (like in quantum mechanics) they might be simply due to the limited resources available to us in the real-world.

3 Real Time Metrics

While there have been discussions on Real Time vs Real Fast at a qualitative level, we believe that a quantitative approach is mandatory for system level comparison - the long history of failed RT-metrics might suggest otherwise though.

We are not claiming that we solved the problem of RT-metrics, but we will give it a shot to introduce two fundamental metrics that allow system comparison.

- Inherent System Randomness
- Timestamp precision

These two metrics are the lower bound constraints for any quantitative statements of higher level metrics (i.e. interrupt response, scheduling jitter or WCET) in complex computing systems.

3.0.1 attempt at a definition

Inherent System Randomness: Non-determinism of a complex HW/SW system due to the limitations of putting any task (SW) into a well defined initial state.

The issue here simply is that, while it is not disputed that HW and SW in principle are deterministic (functionally and temporally) - the actual problem is to determine the initial state of both the hardware (i.e. CPU internal shadow registers, state of execution units, pipelines, caches, external controllers...) and software.

Missusing the concept of the pilot-wave theory [7] the claim is that the systems global state and the next state transition are well defined at all times, but not known by the observer; the initial conditions of the system (HW and SW) are not known accurately, so that from the point of view of the observer, there are uncertenties which can be modeled as random characteristics.

While one can argue that it might be possible to actually determine the initial state of a modern CPU or even the entire system, this ability is not relevant for any practical system and our experiments at de-randomization suggest that achieving a well-defined state at application level is at least not practicable for any real-life application (see below)

3.0.2 Measuring randomness

The first problem to resolve is to demonstrate that a modern CPU actually exhibits inherent randomness - that is randomness that is not triggert by external events like interrupts.

The actual metric proposed for this is the quality values for random bitstreams produced by execution time randomness of a trivial code construct - something like:

```
__inline__ unsigned long long 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;
}
```

If this were run as a tight-loop then patterns in sampling the TSC do emerge (though they still exhibit a high-level of entropy) - if the loop is allowed to run "unknown" code in between by the call to usleep, the randomness of the sampled TSC reaches very high levels, comparable to TRNGs based on background radiation of thermal noise - thus clearly indicating that the OS is "randomizing" access patterns. It should be noted that even when run with disabled interrupts this works just fine.

If the underlying system is deterministic this code will yield a non-random sequence - but as measurements show the output is comparable (if not better) than random generators using background radiation [9], thermal noise [8] or interrupts events [4] to "generate" entropy.

Readers will note that it is technically not feasable that the results of a software RNG - even if sampling the inherent randomnes of the CPU - be of better quality than a geiger counter sampling background radiation - this supprising result can be explained by reviewing the driver used at hotbits.org [10], which uses a serializing instruction befor the call to rdtsc and futther uses a slow peripheral (serial port) as input device - these two factors reduce the randomness of the sampling process and explain why such a lower randomness can be observed.

The second metric (and this still needs some work) is to look at the execution time variance of simple code sequences on the respective hardware. The interest in the behavior of simple code sequences is simply that if these exhibit a non-deterministic behavior timing wise then in a preemptive system like GNU/Linux this will be amplified at the higher levels of the system as small variances in execution times in individual code-paths induce unpredictability of instruction sequence at the CPU level. Thought these variations might seem small the impact on modern CPUs is quite large due to the internal complexity of these CPUs (multiple pipelines, prefetch-units, pipline inter-dependencies, etc.).

To measure this we again utilize a randomness metric but instead of calling on the OS to "randomize" the state of the CPU (the call to usleep(delay)) we put a well defined code sequence before the actually measured code and look at the timing variance of the fixed sequence:

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

The actual code then looks like this:

```
for(i=0;i<32;i++){
   /* this code is a deterministic warmup */
   for(j=0; j< warmup; j++){
      x1=i;
      x2=x1*i;
      x3=x1*i;
      x3--;
      dummy+=x3*x2-(x1/4);
   }
   __asm__ __volatile__("rdtsc\n\t":"=A" (x));
   res|=(((x&bit)>>shift)<<i);
}
```

The code sequence only references local variables, is using a simple subset of instructions (arithmetic instruction available on all systems) and the results are stored in an array that is much smaller than the L1 D cache (we chose 1/8) to minimze caching effects. Further this is run with interrupts disabled. By doing so we can observer the randomness of the CPU it self, and by iterating over the number of warmup_loops used we can seek to "de-randomize" the CPU (which currently seems to be doable only in very specific tuned assembler code cases). Thus what we get from this is the inherent variance of the CPUs execution time for a given simple sequence of instructions - any attempt to rely on a system providing a higher level of determinism seems unrealistic as impact of interrupts and caches will make things even worse.

As actual metrics we propose the use of the evaluation code presented at random.org [11], which tests for

- Entropy
- Arithmetic Mean
- Chi Square
- Monte Carlo Pi calculation
- Serial correlation

A typical run of this on a AMD Hammer (UP) will yield:

```
Entropy = 7.625602/byte.
compres = 4 %.
chi sqr = 71.09.
Arit mean = 119.2637
Mont Car Pi = 3.105882353, err. 1.14 %
Ser. Corel. = 0.041795
```

(Note that only the output format of `ent` was reformatted to allow better script processing - the tests them selves are unaltered).

A verification with the full test-suit from TestU01 [6] convinced us that the results are actually statistically random in nature reliably. Further these test have been done on AMD Sempron, AMD Duron, Intel Core Duo 2, Intel Celleron

Note on other architectures: on MIPS (Loongson 2F) we were simply not able to access the respective register from user-space directly - so this simply measurement method did not work, on PowerPC (405/440) we had similar problems with direct access to the Timebase - so this is X86 only at present - for other architectures kernel level implementations are in the works.

timestamp precision:

Why timestamp precision ? Any decisions made based on time mandate that an event of interest can actually be timestampt precisely - thus the timestamp precision is a lower bounds for any time related decision in a system. Specifically in an RTOS no decision can be more precise than the timestamp capability of the system.

Timestamp precision - dependant on:

- time source resolution
- inherent randomness of the CPU (hardware)

- inherent randomness of the system (software)
- isolation of the time-sampling code

there is no point to claim any bounded jitter on scheduling that is below the inherent randomness of the system it self - this is a hard lower bounds for timestamp precision.

The time source resolution has been increased dramatically over the past decade allowing at least microsecond resolution on most current systems, generally allowing resolutions of one to 10 cycle at CPU frequency (though this depends on specific settings on some systems).

At the same time the complexity of CPUs has increased to a point where the inherent randomness of the CPU and its interdependency on other hardware units (i.e. FSB, north-bridge) makes it useless to increase the time source resolution any further.

At the software level the impact of functionally unrelated code has been improved (notably in RT-preempt) in GNU/Linux providing relatively good code isolation under appropriate configuration.

So timestamp precision in CPU cycles would be one of the proposed base metrics for a RTOS. The timestamp precision of system we measured can be as bad as a few hundred cycles due to the impact of serializing instructions. Claiming scheduling jitter below the timestamp precision technically makes no sense in our opinion.

3.1 Measuring timestamp precision

The actual metric proposed for the timestamp precision is quite trivially, run two calls to consecutive calls to "rdtsc", calculate the difference and search for min/max values.

```
while(n < loops){
    unsigned long long index=0;
    usleep(1);
    hwtime2 = rdtsc();
    hwtime1 = rdtsc();
    jitt=hwtime1-hwtime2;
    index=jitt/scale;
    if(index > GRAPH_SIZE){
            out_of_bounds=1;
    } else {
            graph[policy][index] += 1;
    }
    n++;
}
```

If this is now run at different priorities and scheduling policies one gets an overview of what the lower bounds of any timing decisions at code level will be for the given settings. At the same time this gives a suitable lower bounds for the OS level scheduling jitter. What ever is able to interrupt or intrude the two consecutive rdtsc calls is also able to impact any other instruction (note there is no cpuid instruction used as we are interested in the possible impact of the pipelines) thus even if the OS does better at scheduling the application would not benefit from this simply because the scheduling tests will indicate when the application code got started, but not when the code was able to actually perform the intended operation. If we know how intrusive the OS environment can be then we get a better estimate of the actual jitter of any action code might be taking.

Note that we have not used a serializing instruction to read the TSC (rdtscp or cpuid+rdtsc) as this does not improve the timestamp precision - quite the contrary, serializing instructions have a profound negative impact on timestamp precision and are infact one of the main limitations to timestamp precision as cpuid it self can take up to a few hundred cpu-cycles on multiprocessor systems in the worst case.

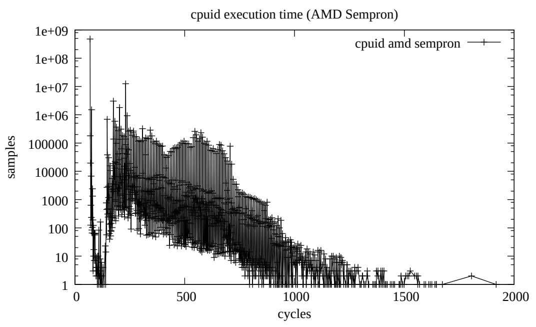

FIGURE 2: *cpuid execution time in cycles (AMD Sempron)*

4 De-randomization

To actually confirm that on of the sources of inherent randomness in a modern computing system can be attributed to the CPU, we investigated the ability to construction code that would not exhibit random execution times. The code in question is very simple code and the constraints needed to achieve this deterministic behavior preclude use of these methods for any real-life code - never the less it is interesting in-sight in the level of randomness of a modern CPU.

Measures taken to "de-randomize" code include:

- warmup loops (to ensure cache hot code/data)
- Fitted into L1 cache or even a single cache line (instruction and data cache)
- only local data involved that fit into a L1 D cache or a single cache line.
- nosmp on multicore boxes
- interrupts disabled
- dynamic cpu frequency scaling disabled

- well selected CPU frequency - not all CPU frequencies allow constant execution times to be attained
- serializing instructions (cpuid)
- manually tuned loop, both length and instruction distribution, to fit pipelines
- simple set of instructions (typically a single one-byte instruction, i.e. "inc

Even with these quite excessive de-randomization efforts which mainly focus on limiting the physical resources involved in the execution of the code to an absolute minimum, even within the CPU, it is not reliably possible to achieve absolutely constant execution times - though the variance can be reduced to roughly 10E-9/10E-10. The CPU in this setup was not using most of its execution units (FP,SSE,complex-ALU,barel shifter) but rather limited to a subset, which tentatively explains why the randomness was reduced.

A side note on serializing instructions - cpuid - does not increase the precision of rdtsc - it only prevent out of order execution - but the penalty of cpuid it self is so high that it actually reduces the precision of timestamp association. On a 1.8GHz AMD Sempron a cpuid causes an overhead of 65 cycles, reducing the variance of rdtsc - but the offset is far larger than the variance - so the net precision is reduced. In fact one of the sources of decreased randomization in the random.org device based on background radiation monitoring might well be the cpuid instruction along with the serial line used for triggering the cpuid rdtsc - this is a bit speculative at this point though as we have not been able to verify these claims due to lack of appropriate equipment.

Further it showed that the actual parameters (i.e. inner and outer loop length CPU-frequency selected) had to be re-adjusted to fit each CPU, we from this conclude that it is not possible to write any meaningful code for a modern CPU that would actually provide constant execution times over even a closely related set of CPUs, and thus de-facto any real-life code exhibits inherent randomness.

5 Conclusion

The maybe most provocative conclusion at this point - modern CPUs are inherently random and a complex general purpose OS on top amplifies this inherent randomness substantially. More work on this is in the pipeline, and we hope to provide more evidence of this speculation in the neer future.

From the current work we have drawn three main practical conclusion are:

- Real Time metrics must be found that can reflect the inherent randomness of modern complex hardware/software systems
- A set of accepted metrics to describe the basic system parameters of inherent randomness and timestamp precision is needed to make systems comparable
- A probabilistic approach to real-time performance to us seems the only meaningful one.

As a starting point we propose the use of:

- timestamp precision in cpu cycles based on a simple test-code that iterates over priorities and considers serializing instruction impact.
- system randomness based on entropy, chi square, Monte Carlo simulation of Pi and serial correlation.

While we were able to find a, in our opinion, convincing demonstrating of the inherent randomness of modern systems by providing a software based random-number generator, we think this is currently not much more than a discussion input and should lead to a widely accepted metric for system randomness to base "high-level" real-time metrics on. The second contribution is the definition of, though trivial, timestamp precision tests. Essentially it is not relevant to which specific phenomena the variance can be attributed, essential is only that these phenomena can't be evaded with acceptable penalty as the de-randomization tests showed and thus any real-time metric will have to take this inherent uncertenty of time-stamps into account.

Practical use of these conclusions are proposed for the initialization of random number pools at boot time of GNU/Linux or for entropy generation of systems that don't have sufficient sources from asynchronous events (typically on embedded systems with only low-bandwidth peripherals). Work on this is under way.

In future works we hope to cover inherent randomness in more depth and find suitable models that allow estimations of execution times for actual real-time systems based on complex hardware/software like GNU/Linux with its real-time extensions on COTS hardware.

References

[1] Matthew Davis & Sameer Niphadkar, *LibMT-PRNG: A Multithreaded Pseudo Random Number Generator* Dr. Dobb's Journal April 20, 2009

[2] Sameer Niphadkar & Matt Davis, *Random Number Generation via Threadding*, December 12 2009

[3] Viega, J. *Practical Random Number Generation in Software* Proceedings of the 19t h Annual Computer Security Applications Conference. December 2003.

[4] Jake Edge, *On entropy and randomness* `http://lwn.net/Articles/261804/`, December 12, 2007

[5] Maurice Herlihy, Nir Shavit *The art of multiprocessor programming*, Morgan Kauffman, February, 2008, ISBN 978-0-12-370591-4

[6] Pierre L'Ecuyer, *TestU01: Testing Random Number Generators*, `www.iro.umontreal.ca/ simardr/testu01/tu01.html`, 2002

[7] Wikipedia,*http://en.wikipedia.org/wiki/Pilot_wave*,2009

[8] random.org, *RANDOM.ORG - Introduction to Randomness and Random Numbers*, `http://www.random.org/randomnes`

[9] HotBits.org, *HotBits: Genuine Random Numbers*, `http://www.fourmilab.ch/hotbits/`

[10] hblogo.zip, *HotBits driver* `http://www.fourmilab.ch/hotbits/source/hblogo.zip`

[11] John Walker, *random.zip* `http://www.fourmilab.ch/random/`

[12] Nicholas Mc Guire, Qingguo Zhou, *Benchmarking - Cache issues*, Proceedings of the 7th Real Time Linux Workshop, Lile 2005