Benchmarking - Cache issues

Nicholas Mc Guire and Qingguo Zhou
Distributed and Embedded Systems Lab
Lanzhou, P.R.China, 730000K
mcguire@lzu.edu.cn

Abstract

The purpose of benchmarking systems is most commonly to alow judgment if a system configuration is suitable for a given RT-spec. Aside from the fact that these RT-specs are most often quite vague, the problem of benchmarking is not well resolved in the currently available hard real-time solutions for Linux.

As a guidance to many of the findings presented here we have adopted the following theme:

Celestial navigation is based on the premise that the Earth is the center of the universe. The premise is wrong, but the navigation works. An incorrect model can be a useful tool.

The issue thus is to validate the models on practical code and not drop to deeply into a theoretical view. In this article we summarize our experience with benchmarking RTLinux and with some partially paradox optimization strategies that we have derived from benchmark results.

1 Benchmarking Problem statement

Any program that is supposed to reliably add a service to a software system requires that it be validated after software development has completed the initial release. Any software that is to rely on available services needs to verify and evaluate its performance in the context of the system it will actually be operated on and verify that it does not adversely affect any of the previously existing services. With system not only the hardware platform but the entire environment of operation is to be considered, including any external inputs to the system (intentional like a keyboard, or non-intentional like network load due to smart OS broadcasting). This testing, generally referred to as benchmarking, is common to all software, but the rules for benchmarking realtime applications has some specifics to it, due to the strong influence on temporal behavior that the environment imposes on such a realtime software system. In the following paragraphs, isolated problem conditions will be analyzed at a very practical level, the goal of this work is not to anticipate a scientific mechanism for analyzing realtime systems, but to guide a practical approach to benchmarking a RTLinux system and especially point out some of the common missinterpretations of results - especially regarding interrupt impact on RT-Linux.

Benchmarking of RT-systems generally will be split into three distinct parts.

- General Purpose OS Layer - Benchmarks of the underlying system services your application relies on in the context of the environment it will operate in (disk read/write, network bandwidth, system call overhead etc.).
- RT-Layer - Testing of the RT-mechanism implemented in the system with respect to its temporal properties (sheduler, interrupt dispatcher, ipc, etc.).
- Application Layer - Analyzing the specific properties of your application with respect to the demands that the anticipated utility, that it should provide, will have (computation time, disk/memory usage,etc.).

Doing a number of industry projects showd that many times one of the three above steps is not performed at all, or in some cases all three of them, and in most cases the tests were either incomplete or performed in a "clean-room" environment being more or less useless. Following the non-existant benchmark results, a lot of speculation on the cause of excessive jitter in the system results in 'system tuning' at a

sometimes very speculative level. This is not to say that these optimizations don't sometimes work, but they can be improved, and especially made platform independant by systemactic analysis of the underlaying problems.

*If one benchmarks an application on a 100Mbit point-to-point connection to your 2 GHz Pentium system on a dedicated network link - you will get (void *)NULL usable results for the behavior of this system on a factory floor with a dozen Microsoft systems happily broadcasting like mad.*

There will be many ideas and concepts about the "right" way to benchmark a system - the solution presented here will not be, and does not claim to be, the absolute truth on RT-benchmarking. We hope to offer a mapping of benchmark artefacts to underlaying mechanisms in a way that allows better interpretation of results and better indentification of causes and thus improved therapy. Feedback on your experience would be most appreciated so any comments/corrections/flames to mcguirelzu.edu.cn - thx !

In this report we will be focused on the second and third part of benchmarking the RT-layer and the RT-applications, which unfortunately are not completely independent of the GPOS nor of the non-RT applications (seen as part of the GPOS layer).

1.1 System load

The influence of the hardware in a system on the systems performance is obviously something that one needs to analyze. Anybody designing a system will take issues like data bandwidth, network bandwidth, peak memory requirements and the like into account. The rules applied for such a decision in a non-realtime system are only of limited use for a realtime system. This is basically due to the fact that a non-realtime system are using optimization strategies to overcome the limitations of hardware in many situations, and resolving the peak resource issue by including fall-back resources that can satisfy peak resource loads at a lower performance. Think of the buffering of disk-writes, or the reordering of TCP-packets in the network layer to prioritize a service over another which may go as far as dropping a low priority packet and waiting for a retransmission of it later or as an example of slow fall-back resources - swap partitions are one well known case. For realtime systems these methods of optimization are not usable as they would break determinism of the tasks, if they are tolerable then that part of the task should be run on the non-realtime side of the system and not in RT-context.

As Linux is targeting best average performance and does not really care about worst case, furthermore there is a slight preference of interactive over non-interactive jobs, the analysis of the hardware related system parameters can be fairly simple if they are quite close to main stream desk-top and server setups. Consider the demand to write 2MB/sec of data to disk, this can easily be mapped to a hardware setup supported by Linux and is able to satisfy this demand, simply by consulting the available data on Linux from the Internet, but nobody will provide a written guarantee that the system will reach this level of throughput under all circumstances. The question of the influence of such an I/O subsystem load may have on the RT-side is harder to answer and will be quite hardware specific. As an example, a system with 32MB of RAM will have to do very high frequency write-cycles to the disk to achieve 2MB/sec disk bandwidth as there is not enough memory available to build extensive buffers for block reordering and caching, if the same system had 256MB RAM the number of Interrupts from the mass-storage subsystem would decrease clearly and thus the influence on the RT-side would decrease. This means that a clean analysis of existing benchmark strategies is necessary, and as we hope to show clearly a redesign.

The main issues with the existing benchmark codes is that they focus on recording specific parameters in RT-context, and it is up to the user to produce 'well-defined' system loads allowing to judge the systems behavior in the target execution environment.

This approach has obvious flaws:

- RT-disruption does not map to Linux system load
- The setup does not permit identifying sources of RT-disruption
- The specifics of the target environment may not lend them selves to triggering accumulated events of relevance
- The specific data-set on which the target environment will operate might not be that well know.
- ...and a few more...

The approach one can commonly find in reports on mailing lists is to load the box to a point where one is confident that such a load case will never happen during actual operation. Typically this is done by

- compiling the kernel with `make -j 30`
- ping flooding with `ping -f`
- running multiple `find /`

So what is wrong with this aproach ?

`make -j 30` does not tell you much as a compiler has a fairly nice balance of CPU and I/O load and code locality is not as bad as one might think, furthermore the granularity of multiple processes in Linux, of 10 milli seconds, leads to a fairly continuous cache load but hardly introduces wild peaks.

`ping -f` is benchmarking what ? It produces a lot of interrupts - but those interrupts go through the interrupt emulation/isolation layer so they cause fairly small RT-related effects, they do cause a substantial I-cache related flush in Linux, once the processing in non-RT starts, but that is an at least partially synchronous disruption. But more importantly `ping -f` is a very wide range, we have found systems (old systems mainly !) that would reach ping a frequency of roughly 5 kHz, and new systems that would hardly reach 500Hz, `ping -f` is a very non-reproducible test and is thus not very well suited for such benchmarks. An even worse problem is that a `ping -f` can isolate the RT-system from the user-space code because the system is kept bussy in kernel-mode, which is a flat address space with RT. One can show that running `find /` alone inflicts higher jitter than running `find /` and `ping -f` in paralell !

`find /` was seen as a nice test, simply because of `find` not being to smart about using buffers and thus causing many disk interrupts. Obviously the results found this way will be very dependant on the underlying mass-storage subsystem and the filesystem layer, but one of the greatest variables found in the `find` test turns out to be the used display, notably frame-buffer device's and the used X-server. Therefore these tests again don't lend them selves to a well reproducible system load that would allow a comparison of systems. To see the effect simply run it as `find / >/dev/null 2>&1` which is obviously producing the same disk-load but no terminal I/O - down goes the jitter or RT-Linux . A further problem is that this is dramaticallz dependant on the harddisk settings (i.e dma enabled or not shown below)

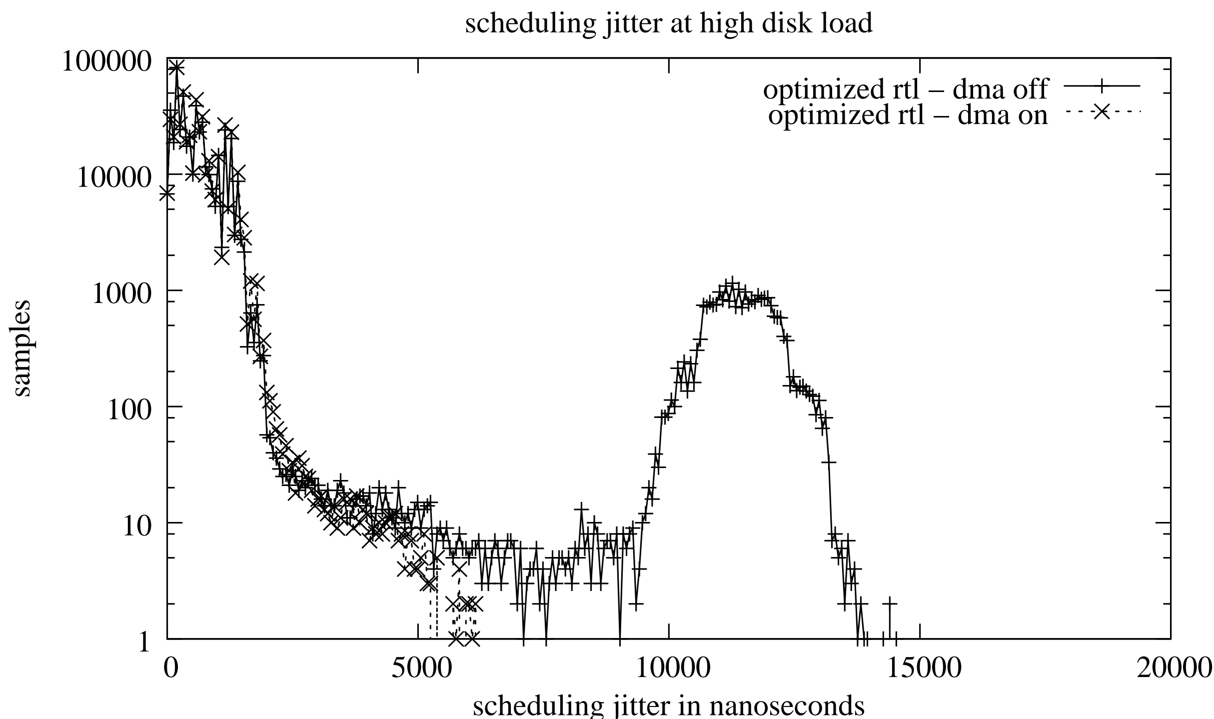

FIGURE 1: *caption*

In general all three noted tests, and the criticisms holds for many of the other suggested 'system loads', are platform specific and show a low reproducibility. And even if they were reproducable what would they say about the specific load caused by an FFT in user-space ?

One of the prime goals of this benchmarking approach thus was to find a reasonable set of applications that can be used to provide well reproducible system loads.

1.2 Test codes

The above criticism mandates that a cleaner and more reproducible approach be taken. Our design guidelines were as follows. With cleaner we mean not only the code induced effects on the system but also the interpretability of data.

1.2.1 RT test-applications

An analysis of existing test cases for RTAI and RTLinux have shown that they all, deliberately or not, enforce a fairly large data locality. This was basically done by having a small number of local variables that were used to register min/max values of some timing and then report these values over a loop period via a zero-copy IPC mechanism (i.e. rtFIFOs). Thus the numbers obtained by these tests are fairly optimistic and can hardly be achieved by real life applications (Note that RTAI uses the identical test-cases that RTLinux uses).

To eliminate these issue a comparatively simple test with a deliberately low data locality was introduced (`jitt_stat.c` - see below). A further problem of the default test cases used in RTAI and RTLinux is that these tests report values that are no longer suitable to perform statistical operations and corelation analysis - we belive that this is at the core of some common misunderstandings with respect to performance limitations of the X86/PPC platform (discussed later on in the section on TLB and cache benchmarks). A main issue for our design thus was to allow data sets produced to be suitable for statistic methods - basically this means, at least record each value. Ideally temporal corelation would also be available, mandating the recording to be fully ordered, but all attempts at this have shown a tool large impact on the systems performance to be usable, a possible solution would be an external recording (logic-analyzer or the like) (TODO: design software independent test-cases).

1.2.2 The User-space load applications

Based on the criticism of exiting benchmarking practice we have designed simple but hopefully sufficiently effective load-apps that target a specific hardware unit (or at least focus on a specific hardware unit). To validate the test applications oprofile-0.7.1 and oprofile-0.8 were used. Further direct reading of performance monitor registers (MSRs) was used, though it showed that a very fine grain reading of PMCs is problematic as it inflicts a noticeable penalty (i.e. reading MSRs/PMCs in core functions of RTLinux). Never the less we think that a integration of PMC reading in the RTL_Tracer could yield better understanding of hardware induced effects.

1.3 Sensitivity

Error handling needs to focus on two distinct cases, the non-RT and the RT side of the system.

If syslogd decides to have a party logging 2MB/sec to /var/log/messages then you probably will end up having far more interrupts from the storage device than you had taken into account during the system analysis phase.

Error handling paths in RT-context may inflict large delays due to cache/TLB effects if they exhibit low code/data locality (at least for the Linux kernel this is the case - no particular care is given to these rare cases, which is legitimate for a non-RT system, for a RT-system error handling could result in an escalation of temporal errors).

1.3.1 Benchmarking sensitivity

This leads to a problem that is more or less impossible to cleanly solve - one can not take all potential sources of disruption into account. So the proposed strategy is to annalize the sensitivity of individual hardware artefacts and then come to a severity tagging for each of these potential hardware artefacts. This is somewhat easier to do as one can fairly easy produce well reproducible disruptions (interrupts,TLB flushing,cache flushing, BTB exhaustion, etc.).

What preliminary tests have shown until now is that the effect of hardware artefacts in some cases can be distinct enough that events can be assigned to these artefacts provided a minimum set of timing data is logged - clearly this logging causes an overhead but as the tracer implementation shows: the temporal overhead and distortion as well as the resource requirements can be kept fairly low.

1.4 Applying benchmarks

Before you can go and actually do reasonable benchmarks you need to designate the bottlenecks of interest and roughly sort them by severity. So interpreting benchmarks beyond pure "my box has 20 bogmips more than your box", we need to introduce the bottlenecks first.

The good news is that once you get the bottlenecks straight, code optimization kind of becomes a natural issue. And that is one of the main objectives of benchmarking - to guide you on hardware descisions and code optimization strategies after a bottleneck was indentified as being critical. The second objective of benchmarking is to record an initial performance profile of a system to allow detection of potentially harmful developments at runtime - though

this second objective is of importance to applied embedded and RT systems it is not covered here as this is out side the scope of this study, noted here only as the methods will not be substantially different than described here (although I know of no project to date that actually bothered to implement system runtime monitoring at this level...)

We end up with a fairly short list of low level bottlenecks:

- system busses
- memory hardware settings (memory timings)
- memory hierarchy (L1, L2, Write-combine buffer)
- memory management unit (TLB, pgd).
- branch penalty (BTB buffer and prediction 'algorithm')

Some might be supprised not to find interrupts here - our findings are that interrupts are really not that much of an issue, in fact one can isolate RTLinux from interrupts in a way that a ping flood (which de-facto is high interrupt load with close to no data load) has no substantial effect on the RT-system any more.

1.4.1 code design

One of the main problems with benchmark code design is that in RT-systems we are interested in worst case events, but due to the steadily increasing resources in general purpose architectures, exhausting a specific unit may not be that easy and even less reproducible. Further more most tests do not isolate well enough that they only test a single unit, but due to side effects will test multiple units and unit interaction with a variance of unit-load distribution only.

The increase of resource, i.e. cache sizes, BTB size, and the fact that in some cases execution streams will NOT utilize these units (i.e. tight loops not querying the BTB at all, pre-fetch mechanisms, and write reorder buffers) mandates that benchmark code be reviewed at the assembler level and not only at the C-source level. It may well happen that gcc is too smart when attempting to write bad code deliberately - in fact our first attempts to write naively bad code resulted in GCC optimizing almost everything away that we did to hurt performance !

Further it should be kept in mind that average case optimization often implies rare cases of performance penalty taken into account (fast-path/slow-path concepts), this means for RT-systems that one can encounter seemingly paradox effects of "bad-code" performing better when it comes to worst case than well optimized code. This is especially true with respect to branch optimization based on profiling data (GCCs -fprofile-generate and -fprofile-use), which yields best average throughput (in most cases) but will not optimize the worst case, in fact generally optimizing for throughput implies inflicting higher worst case response times.Even optimization for average throughput by means of profile data feedback yields case-specific optimizations and only is efficient if the system/application-load profile during profile generation reflects the actual runtime situation. Even if this might be an over statement, profiling data will never handle the worst case, the wors-case is a rare case, profiling catches the average case only and thus is not too helpfull for hart realtime systems.

A further critical issue that popped up, and it took us quite some time to detect this, is that the RT-subsystem (RTLinux/Pro,RTLinux/GPL,RTAI,ADEOS) is quite small so having multiple threads in RT-context that exhibit a very high RT-load and permit a small share to be utilized by Linux only, will show some "insolation" effects. What this means is that the cache lines are filed with core RT-routines, and as long as one stays in RT-context one hardly will flush the i-cache or the BTB, thus exhibiting good performance. One symptom of this isolation effect is that single low frequency RT-tasks will perform worst than multiple RT-tasks or high frequency RT-tasks. Obviously this means that one needs quite detailed information about the target scenario or one must sweep the entire spectrum. This can be seen in the following figures in the sweep that shows a decreasing worst case jitter up to about 50kHz RT-task frequency. The slow task will find all its cache lines flushed - thus a clear increase in worst case jitter. For pure interrupt load this effect is obviously much waeker, but never the less visible.

This sweep was done up to 200kHz with a shift of jitter to hard peaks of low jitter at 2-3us at high frequencies - but as at this frequency hard realtime can not be guaranteed reliably any more as one is already missing deadlines (though in rare cases only) we did not pursue these tests any further. The interesting part is to see that at very high frequencies there is a self-isolation effect with the non-rt system simply no longer able to inflict any harm on the RT-application. This effect is visible both for interrupt load and memory subsystem load.

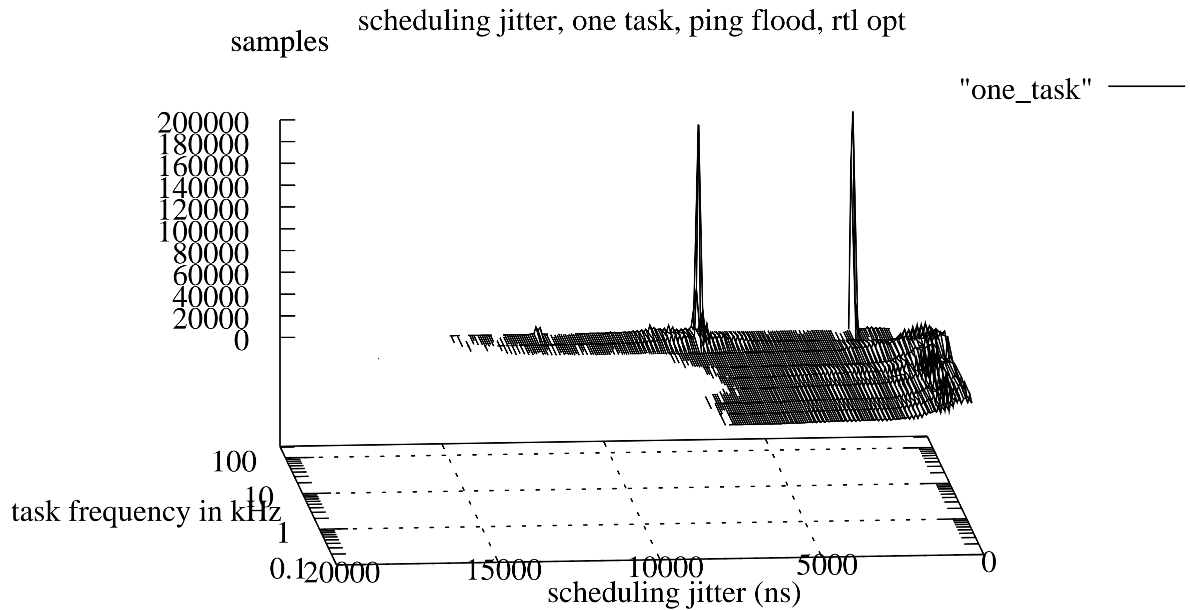

FIGURE 2: *caption*

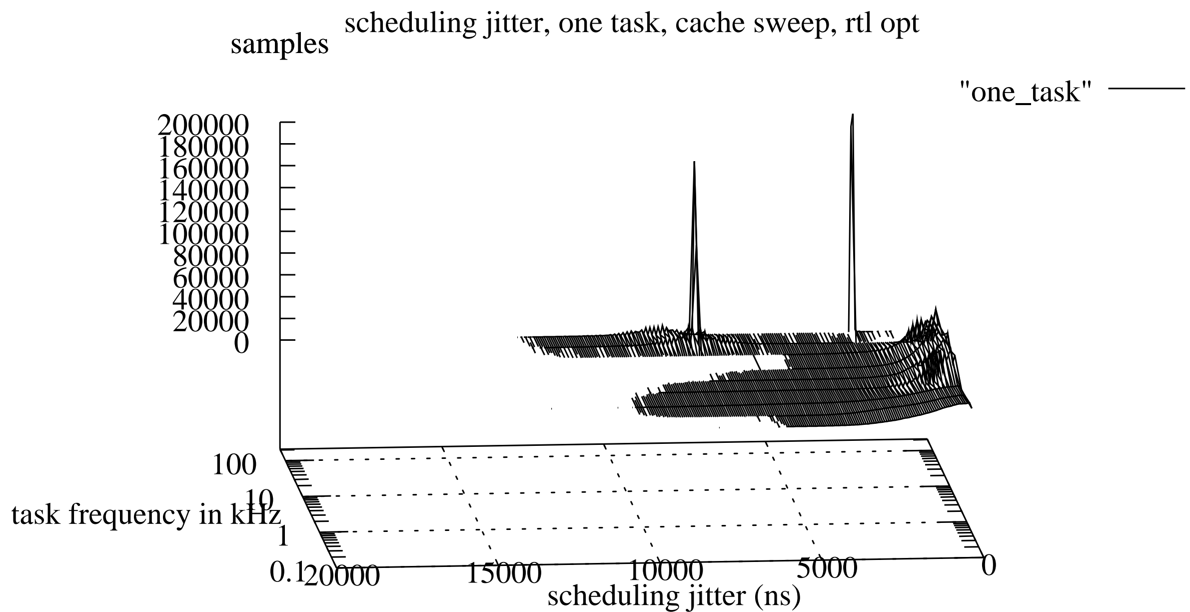

FIGURE 3: *caption*

A further isolation effect can be seen in the standard tests of RTLinx and RTAI that offer to run two tasks for benchmarking task-switch times. The problem is that the low priority RT-task that is to be preempted by the high-priority RT-task is causing an unrealistic code/data loality as this task does not actually do anything but run a tight loop for enough time to be preempted. this effect can be see by the worst case scheduling jitter of this test running only the high priority task and non-RT tasks is higher than if both RT-tasks run.

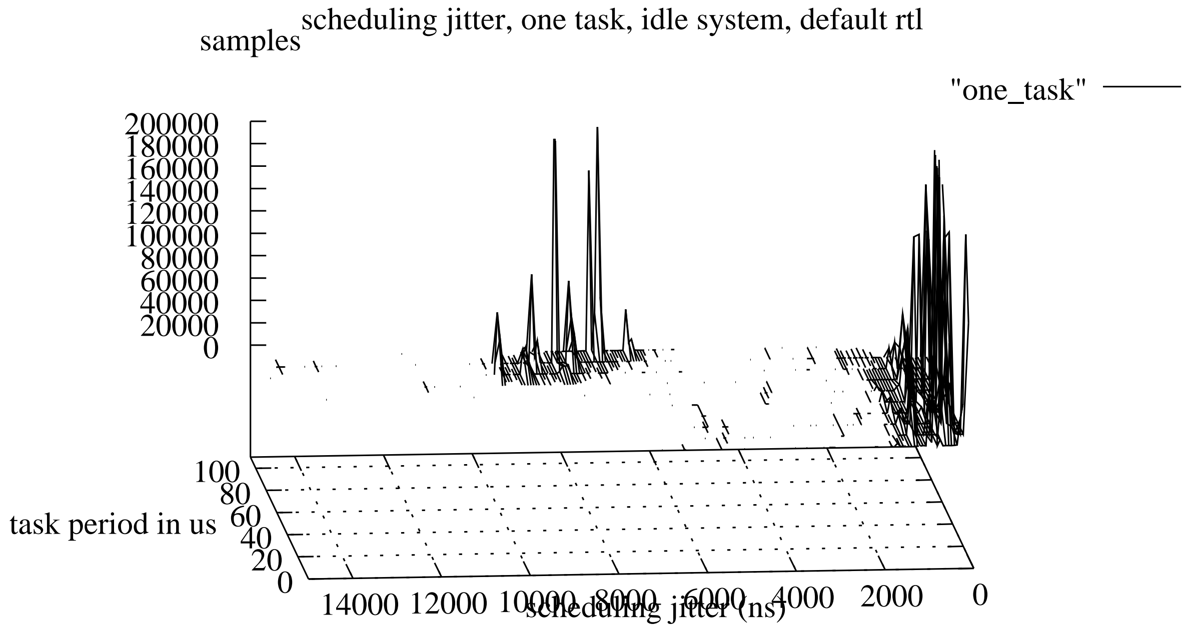

FIGURE 4: *caption*

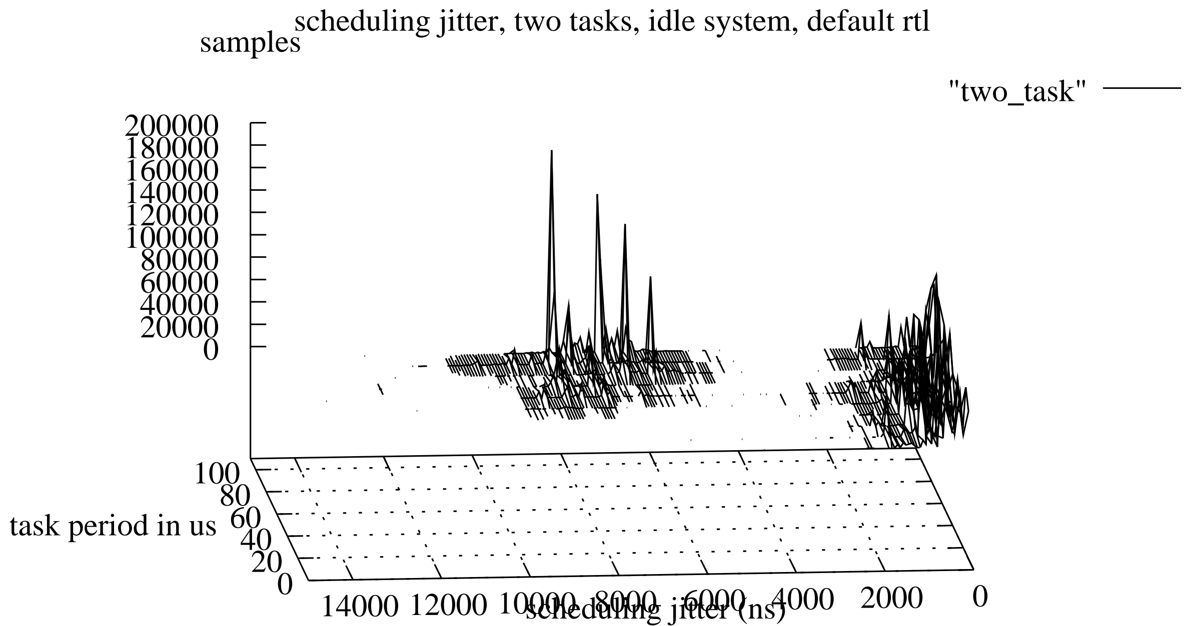

FIGURE 5: *caption*

When benchmarking RT-systems one should have to distinct benchmark sets

- RT-RT benchmarks
- RT-non-RT benchmarks

The first set will produce large RT-loads and then measure a parameter of interest in one of the threads, the second basically should be running one RT-task only and vary the non-RT loads. Naturally one can construct a number of variations of these setups, but we don't belive that is too useful, as RT-benchmarking is only a check for "can the system NOT do it" - it will most likely not be possible to say if the system can definitely do the job - not unless the exact task-set is known!

1.5 User space isolation effects

A further case of isolation is user-space processes putting a specific load on the non-RT system and thus preventing Linux from causing jitter in the RT-subsystem. Even during regurlar benchmarks one can see that the minimum values of jitter tests are reduced, indicating that a user-space load that results in good data/code locality will result in false-positive evaluation. A ping-flood for instance, will on a low end platfrom, result in the non-RT side being permanenntly busy in the ping related code section in the kernel, and thus the user-space simply more or less stalls, resulting in a very unrealistic load profile and thus false results. An extrem case of isolation can be seen by running a trivial:

```
int main(){
   while(1);
}
```

resulting in a pure CPU load but absolutely no memory subsystem or I/O subsystem contention and thus the RT-subsystem performs much better (both average and worst case scheduling jitter) during such a "high" user-space load.

2 RT-Linux related cache issues

In this section we will introduce the main findings with respect to RT systems reaction to cache subsystem artefacts. Aside from analyzing the systems and validating tests with profiling data, we have fed back the findings into a RTLinux/GPL source tree and validated that the so optimized system actually performs better. At time of writing this improvement is in the range of worst case reduction yielding 60% to 65% and we are not sure if we are at the limit yet. On the other hand we found that the optimization is only in some parts generic enough to be applied for all platforms, many of the low level optimizations are specific to a particular CPU/memory-subsystem setup mandating per-setup optimizations.

2.1 Quick summary

- BIOS settings related to memory subsystem
  - cache disabling is ignored once the kernel boots (booting is slower)
  - memory bus settings are not ignored but influence Linux
- MTRR issues
  - MTRR may be relevant for inactive devices (X86 specific)
  - write-through settings can de-stabilize the systems (not really clear why !)
  - MTRR settings in Linux most likely incorrect (2.4.X kernel)
  - fix up of MTRR code improved system responsiveness substantially
- PCI Bus
  - PCI bus settings (MAX_LAT,MIN_GNT)
  - L2-insolation is sufficient to make PCI settings negligible for common devices (there might be exceptions i.e. HSD,Video-processing).
- Write combine issues:
  - kernel does not use write combined (no RT-effects)
  - user-space RT-may well be impacted by write combined settings
  - most likely disabling them will reduce latency (though impact throughput)
  - I/O writes are not buffered in the write buffer only memory writes are .
- TLB management (flush_tlb/flush_tlb_all/flush_tlb_one)
  - SH4 tlb management instruction (loadtlb)
  - ia32 and PPC have very limited (course grain) TLB management functions only
- Cache management functions
  - ia32/PPC de-facto have no cache management functions
  - some cache related functions appear as 'side-effects' (MTRR setting)
  - P4 clflush assembler instruction (not in use in 2.4)
  - SSE2/MMX/Altivec pre-load instructions (currently not used in RT-Linux variants but they are used in the kernel i.e. memcpy)
  - Branch penalty
  - cache penalty of branches can be influenced via (likely/unlikely in core modules)
  - GCCs (__builtin_expect()) cache effects of instruction ordering can have a negative impact if incorrectly used.

- predictable if - well defined prediction state (not taken)

- Misc issues
  - Arch selection while compiling (basically turning off optimizations) inefficient - selectively disable MMX/Altivec and atomic SIMD instructions
  - Compiler (2.95 vs 3.2.2) negligible to irrelevant
  - Unused devices should not be present in the system as some devices even if seemingly unused may have degrading effects (notably USB).

2.2 Critical MM related functions for RT-systems

There are not very many places in the Linux kernel where the cache subsystem is directly impacted on - the actual cache management API is de-facto not in use on ia32 and PPC. Some issues where side effects may arise and that would be hard to track down (and in fact catastrophic if they happen during RT-operations !).

- change_page_attr -¿ wbinvl - flushes the entire cache resulting in potential CPU stalls of hundreds of microseconds.!

- MTRR related functions at runtime

- wbinvl is a serializing instruction - long lock on SMP possible

- update of MTRR with WRMSR flushes the TLB

At this point (2.4.X kernels) it is sufficient to make sure that no MTRR settings are changed during RT-operations. The page_attr are currently not updated at runtime (at least we have not found any such kernel path - but admittedly it is hard to exclude this at present (TODO - verify this))

2.2.1 Cache Management functions in Linux

Most Linux kernel cache manipulation functions are unfortunately not really an issue.

List of functions from `include/asm-i386/pgtable.h`

```
/* Caches aren't brain-dead on the Intel. */
#define flush_cache_all()                       do { } while (0)
#define flush_cache_mm(mm)                      do { } while (0)
#define flush_cache_range(mm, start, end)       do { } while (0)
#define flush_cache_page(vma, vmaddr)           do { } while (0)
#define flush_page_to_ram(page)                 do { } while (0)
#define flush_dcache_page(page)                 do { } while (0)
#define flush_icache_range(start, end)          do { } while (0)
#define flush_icache_page(vma,pg)               do { } while (0)
#define flush_icache_user_range(vma,pg,adr,len) do { } while (0)
```

unfortunately they are thus quite brain dead when it comes to predictability as there are no management functions available in ia32. PII++ permits selective control of L2 caches including. loading/flushing L2 lines via MSR - though there seems to be no implementation of this available (at least not in Linux).

On PPC its a bit better (pgtable.h):

```
#define flush_cache_all()               do { } while (0)
#define flush_cache_mm(mm)              do { } while (0)
#define flush_cache_range(mm, a, b)     do { } while (0)
#define flush_cache_page(vma, p)        do { } while (0)
#define flush_page_to_ram(page)         do { } while (0)
```

The available functions are (implemented in arch/ppc/kernel/misc.S):

```
flush_icache_user_range
flush_icache_range
__flush_dcache_icache
flush_dcache_page
flush_icache_page
```

Unfortunately the implementation is quite lengthy as there is no simple low level support for these functions - so the actual advantage is not that clear. Also on many PPC (notably 8xx) these functions are quite heavily in use due to the CPM coprocessor not allowing bus-snooping and thus mandating quite a few flush operations (bad for RT and RT on 8xx does not look too good, although we are only speculating that this is a main issue at this point).

2.3 MTRR settings

As of 2.4.22 it looks like the MTRR settings are not properly set up - actually flipping the MTRR bits on improved the system performance substantially.

The code sequence used for this on PPro/PII++ is as follows.

```
pthread_spin_lock (&cr0_lock);
__asm__ __volatile__(\
        "movl $0x2FF,%%ecx\n\t" \
        "rdmsr\n\t" \
        "orl $0x800,%%eax\n\t" /* turn on MTRR again */ \
        "wrmsr\n\t" \
        "movl %%cr0,%%eax\n\t" \
        "andl $0x9fffffff,%%eax\n\t" \
        "movl %%eax,%%cr0\n\t" \
        "wbinvd\n\t" \
        :::"eax","ecx");
pthread_spin_unlock (&cr0_lock);
```

The fact that this turning on of MTRR improved the performance in a substantial way indicates that the MTRR management in the kernel is atleast partially incorect - it also should be noted though that MTRR enabled systems potentially have a RT-related risk as MTRR manipulation requires flushing of the entire cache. Therefor MTRR usage mandates that RT systems set up the MTRR at system initialization and don't modify settings at runtime. For the test-codes this MTRR setup is done in the benchmark codes simply as to allow varying configurations without requireing recompilation of the kernel or rebooting the system.

It also should be noted that the systems with cache disabled (via MTRR settings) are extremly slow and thus test-codes must be launched with frequency settings roughly two orders smaller than on normal systems or the systems will overload and thus lock up (more than 100

The most notable result of cahce-disable tests was that the jitter in the system dramatically increases in cache-disabled systems due to the lack of bus-isolation (L2) and due to the inefficient bus-mutiplexing (L1 I/D) towards the CPU. Naturally disabling the cache in the BIOS will NOT due the trick as Linux, once booted, ignores the BIOS. This was also validated that the cache enable/disable in the BIOS is ignored. Note though that this is not true for timing settings in the BIOS - these due influence Linux as one would expect !

2.4 Bus-isolation - Unified L2-cache

The L2 caches operate in a speed range of roughly 10-20ns the primary function aside from memory-speed (cost reduction) issues by allowing large (slow) RAM to be used with small (fast) cache-ram is to ensure a proper bus isolation of the CPU's memory access (thus DMA will not cause CPU stalls if data/instructions are available in L2 - which is the case on 486 that has no L2 yet) This bus-isolation is the primary functionality of interest in RT-systems - thus we focused on validation the system in this respect.

2.4.1 Benchmarking bus-isolation:

The PCI bus has a fairly elaborate management interface that is fully available in Linux. The three values of interest to use are MIN_GNT (minimum grant) MAX_LAT (maximum latency) and LATENCY_TIMER.

- Minimum grant (MIN_GNT): the time a device may be expected to lock the buss
- Maximum latency (MAX_LAT): maximum wait time for the device
- Latency timeer (LATENCY_TIMER): the latency timer - after expireing the devcie should release the buss

run 1:

```
MIN_GNT=1 - lock the buss as short as posible
MAX_LAT=99 - wait as long as posible
LATENCY_TIMER=1 - give the bus back as fast as posible
jitt_stat.o period=25000 bperiod=0 cache\_mode=2
```

run 2:

```
MIN_GNT=99 - lock the buss as long as posible
MAX_LAT=1 - request the bus as fast as opsible
LATENCY_TIMER=99 - delay giving it back as long as posible
jitt_stat.o period=25000 bperiod=0 cache\_mode=2
```

If both runs show the same or similar results on the idle system then the bus isolation is asumed to be good. the uncirtenty left though is that some PCI cards simply don't care about PCI settings so the tests may deliver false positive results if the results are the same - indicating good isolation - if all devices simply ignored the settings (note though that this would indicate violation of the PCI specification!).

The rational behind the idle system and the single task is to ensure that a "clean" system in this constelation would show no substantial peaks above a few microseconds - allowsing to see event a low number of delay events that might occure.

Note that the frequency of the rt-task may not in it selfe be an overload resulting in false negative results - a save setting seems to be a single rt-task running at 10kHz (300MHz Celeron - 2GHz AMD-XP tested).

PII write buffers (4 write buffres length 4) can reduce effect of write misses if write instructions that expect misses are no longer than 4 writes.

2.4.2 Arch notes

Intels are using MCHs that isolate the PCI bus well from the RAM/memory-bus but in recent (P4) variants the AGP bus aswell as dedicated devices (i.e. Gbit Ethernet are being serviced directly from the MCH - this is most likely problematic (though we did not verify this as we don't have any such devices available at present).

2.5 Memory Bus mutliplexing L1-cache

L1 cahces operate in the speed range of 1-10ns and provide three core functions

- isolation against write combine buffer
- permit concurent L2 update and L1 access
- provide memory bus-mutliplexing to the CPU data/intstructions subunits.

The first function can be influenced by the MTRR settings in systems supporting MTRR (PPro/II++), the second function is a "pure" hardware artefact and not realy of relevance to software optimization issues. The third is of relevance in so far as especially the branche prediction unit directly depends on L1/I-cache - systems with larger L1 caches (AMD 64kB) show a clearly less sensitive behavior than systems with small L1/I-cahces to branch miss-prediction (P4-Celeron 12kB,PII 16kB).

The predictable if funciton we have experimented with targets this L1 I-cache issues as folows.

```
 if(condition){
body;
 }
```

is replaced by:

```
 if(unlikely(!condition){
;
 }else{
body;
 }
```

the unlikely negated condition servers the purpose of ensuring that the "not-taken" branch is the default and inlined at the current code position gcc gurartees to put the likely path in the continguous code sequence and will move the unlikely code to the end of the current function. The empty if statement is droped by gcc (-O2) and the conditional jump address is set to the instruction following the body (gcc sometimes inserts a nop).

thus we receive:

```
        .file   "if.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "%d"
.LC1:
        .string "1"
        .text
        .p2align 4,,15
.globl main
        .type   main,@function
main:
        pushl   %ebp
```

```
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $8, %esp        :frame setup
        leal    -4(%ebp), %eax
        pushl   %eax
        pushl   $.LC0
        call    scanf
        addl    $16, %esp       :branch start
        cmpl    $1, -4(%ebp)    :condition
        je      .L3             :branch on equality
.L2:
        movl    %ebp, %esp      :not taken path (cleanup)
        xorl    %eax, %eax
        popl    %ebp
        ret
        .p2align 4,,15
.L3:
        subl    $12, %esp       :taken path
        pushl   $.LC1
        call    puts
        addl    $16, %esp
        jmp     .L2             :cleanup
.Lfe1:
        .size   main,.Lfe1-main
        .ident  "GCC: (GNU) 3.2.3"
```

if body is at the end of the current function and will impact on the performance du to cache misses (CPU stall while loading the cache lines corresponding to the appended "body".

```
#include <stdio.h>

main(){
        int i;
        scanf("%d",&i);
        if(i==1){
                printf("1\n");
        }
        return 0;
}
```

```
        cmpl    $1, -4(%ebp) :condition
        jne     .L2          :inverted not taken
        subl    $12, %esp    :body
        pushl   $.LC1
        call    puts
        addl    $16, %esp
.L2:
        movl    %ebp, %esp   :cleanup / not taken code
        xorl    %eax, %eax
        popl    %ebp
        ret
```

This improves the cache behavior for frequently taken conditional code. The obvious opposite "unlikely" behavior is to deliberately push the code of the not taken branch to the end of the function , thus increasing the probability of having code that will be actually needed in the cache line all ready.

2.5.1 GCC Branch related cache control attributes

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

The above defines correspond to the Linux kernel likely/unlikely macros used in the kernel code to represent the __builtin_expect attributes in GCC. the intention of these macros is to optimize the cache behavior of branches by reordering the above assembler output as follows:

2.5.2 Problems with likely/unlikely

GCC's intention are unfortunately not quite mapped to the reality of branch prediction algorithms. The branch prediction unit in the CPUs instruction unit maintains a BTB (Branch Trace Buffer or Branch Target Buffer (found both definitions - don't really know which one is the authentic one), and this BTB consists of a address list together with a branch "history"

```
state 0 - guess not taken
  if taken state ++
  else state = 0
state 1 - guess not taken
  if taken state ++
  else state --
state 2 - guess taken
  if taken state ++
  else state --
state 3 - guess taken
  if taken state ++
  else state = 3
```

Thus the optimization introduced by GCC fails in two cases:

- The branch is new to the BTB - thus guessed not taken and the cache line corresponding to the not taken code is requested any way.
- The branch is known in the BTB but is not yet in state 2++

The second condition can be seen as a "startup" problem and is not really an issue - the first condition is the problem that needs to be addressed - it will be encountered if:

- the branch was really not yet taken (startup)
- the branch was kicked out of the BTB due to too many branches being processed before revisiting the branch address.

The second case is quite common on larger systems and even on embedded systems it will happen as the kernel paths them selves will easily fill the currently typical BTBs of 1k/2k. The consequence of a branch miss-prediction is a CPU stall - if the code is not already in L1 cache. In case the instruction must be fetched from main memory the stall can be substantially - note that typical main memory has latency in the range of 100-200ns and with L2 cache line sizes of 32/64bytes that must be loaded (even if instruction bypassing is available and the stall would hit the second instruction not the first) the stall will easily amount to microsecond order jitter.

The immediate impact on benchmarking here is that for RT-systems this problem DOES NOT occurs with high-frequency tasks as these may well never fall out of the BTB, but impacts on low-frequency tasks. It also should be noted that very tight loops (i.e. spinn-locks) don't use the BTB at all but use the instruction pool to reference the last taken value - thus for very high frequency conditional code (anything hat will fit into the pipeline) no BTB related effects will be found.

To allow a clean benchmarking of BTB related effects we have "designed" brute force BTB flushers as follows:

```
/* unlikely false 2048 */
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)

main(){
    int i;
    long long loop = 0xFFFFFFFFFFFFFFFLL;
    int j,k,l,m,n,o,p,q;
    j=k=l=m=n=o=p=q=0;
    while(loop--){
        for(i=0;i<7;i++){
            if(unlikely(i!=0)){j++;} /* 1 */
            if(unlikely(i!=1)){k++;} /* 2 */
            if(unlikely(i!=2)){l++;} /* 3 */
   ....
if(unlikely(i!=1)){q++;} /* 2048 */
            }
    }
    printf("%d,%d,%d,%d,%d,%d,%d,%d\n",
        j,k,l,m,n,o,p,q);
    return 0;
}
```

With variations for likely (false/true) and unlikely (false/true) the code actually must be 2048 (or how many ever) lines of ifs ! - the 8 variables used were chosen to ensure that gcc can't assign registers to all variables - thus falsely improving D-cache characteristics - and the concluding printf is needed to prevent GCC from optimizing the entire code away as it would detect that the variables are not used and thus don't need processing.

With this code prototype tests were run with 32,64,128....8192 branches of varying type and scheduling jitter was measured (see BTB_3d_plot). The results show that a miss-predicted if(likely()) is fairly uncritical as the code will occupy a cache line but other than that there is little impact if the falsely marked body of the if is not very large (in which the cache miss-loads due impact on the system). in the case of incorrect unlikely the effect is more dramatic (even for a very short body!) as the body will be moved to the end of the function and thus very bad code-locality (spatial and temporal) is introduced.

2.6 TLB Management

The TLB (Translation lookaside Buffer) is the cache for the MMUs translation of virtual to physical addresses. In most (all?) current CPUs the MMU is integrated in the CPU. TLB misses will cause CPU-stalls, due to the small size of the TLB (32pages) the likely hood of such stalls is fairly large - notably due to the low code-locality of the Linux kernel - to exaggerate a bit - the interrupt path through the Linux kernel with a interrupt code from a runtime loaded module will come very close to a TLB flush (a quick ksyms -a will reveal the low code locality !). Therefor a prime conclusion for a RT-system is that statically linking ALL kernel modules at compile time that are providing hardware management is preferable - most likely this can be extended to non hardware related modules as well (FS,etc. - but not checked yet). It should also be noted that TLB issues impact on slow running RT-tasks more than on high frequency tasks, as the probability of the TLB entry for the RT-task to be flushed in a low frequency task is very high (although on a mildly loaded system this is only of statistic relevance - the worst case will not be much influenced).

As X86 provides no explicit TLB loading/flushing assembler instructions (note that it should be possible to load TLB entries via MTRR and the i386 (!) had the ability to load/read TLB entries via the debug registers) only a indirect method of invalidating specific TLB entries (invlpg) or the entire TLB via cr3/cr4 is available.

This limited possibility of manipulating the TLB never the less allows to optimize the system as free TLBs allow quicker loading of new entries than is required if a TLB entry must be flushed first. This to us was a bit surprising and we have not found sufficiently detailed docs on the TLB/MMU management states to explain these effects in detail. The summary of our experimental findings is as follows:

All systems with 32 page TLB:

```
L1 64K+64K
L2 64K unified  -> 1 x invlpg im sched
                 + 1 x invlpg in interrupt entry = lowest latency
L1 64K+64K
L2 256 unified -> 2 x invlpg im sched
                  + flush_tlb() interrupt entry = lowest latency

L1 16K+16K
L2 512K unified -> 4-5 x invlpg im sched
             + conditional flush_tlb im sched
             + flush_tlb_all() interrupt entry = lowest latency
             + additional invlpg flush-points in core rpt-routines
               (clock_nanosleep etc.)
```

The key issue here seems to be that the larger the cache is the more sensitive the system reacts to TLB misses - it is quite irrespective of I or D TLB misses (at least we were not able to measure a relevant impact on increasing D-TLB load - I-TLB load seems to be "sufficiently" high in Linux any way so our I-cache flusher was not able to really push the I-TLB misses up (need to redesign it - not done yet)).

2.6.1 PII speculations on TLBs

plot 9 shows a closes 1/X behavior indicating that the TLB related effects are close to eliminated and that the remaining artifacts are truly related to cache misses (see HPCA)

3 Test codes

Aside from the BTB flush code noted above that was implemented for 32-8192 branches (likely(true/false) unlikely(true/false)) we also implemented a dcache sweeper, and a icache sweep. Both basically do nothing else but aggressively step through a large memory area with variable stride to show cache influence

Not too surprising a stride 1byte larger than the cache line size impacted on the system heavily, stride 1 was not dramatic (as could be expected). The sweeper was the application that profited most from the TLB flush points - that is the RT-subsystem improved (reduced) scheduling jitter significantly by introducing TLB flush points in RT-entry paths (scheduler and interrupt-interception - both in RTAI/RTHAL and RTLinux/GPL - it can though be expected that this works fine for RTLinux/Pro and for ADEOS as in all four hard RT implementation the identical low level mechanisms are used (RTAI == RTLinux-1.X, ADEOS == RTLinux-2.X, RTLinux/Pro == RTLinux-3.X)(pleas no flames for this - but the core interrupt emulation in ADEOS really is very very similar to RTLinux-2.X, although

it should be noted that a very large amount of work was done to utilize this interface in a much more elaborate manner and ADEOS also introduced additional features that were not present in any of the RTLinux variants.)

Note on static code - statically linked code exhibits a very bad code locality (spatial locality) and due to the lack of shared objects also shows lower temporal code locality - thus statically linked executables should be considered deprecated for RT-systems. Naturally the impact of statically linked code on the RT-subsystem could also be improved by flush-points.

3.0.2 sweep.c

data cache / data-TLB sweeper

allocate a large memory chunk and step through it in a non-cache line aligned manner - never reusing data. the core simply is:

```
while(1){
    for(i=0;i<array_size-stride;i+=stride){
        array[i] = i+array[array_size-i];
    }
}
```

3.0.3 icache_sweep.c

icache sweep via array of functions that is stepped through in a non-linear fashion to maximize instruction cache/TLB misses. the icache flushers showed little impact (less than large statically linked apps). The icache sweeper is similar to the d-cache sweeper simply using an array of unique functions. The core code again is:

```
while(1){
   for(i=0;i<NUM_FUNC;i++){
       functions[i]();
       functions[NUM_FUNC-i-1]();
   }
}
```

3.0.4 btb*NUM.c

Branch trace buffer flusher - by selecting a very large value of if statements being processed in an endless loop the BTB de-facto is trashed completely. This code shows a very constant behavior up to the size of the BTB and then a system performance breakdown. The core code is listed above (see section on branch traces). One of the paradox results is that code trying to optimize BTB issues by utilizing likely and unlikely macros will backfire if the BTB ever is exhausted, tests with code to delibarately exhaust the BTB, show that one can not improve worst case execution time with the help of gcc's __builtin_expect.

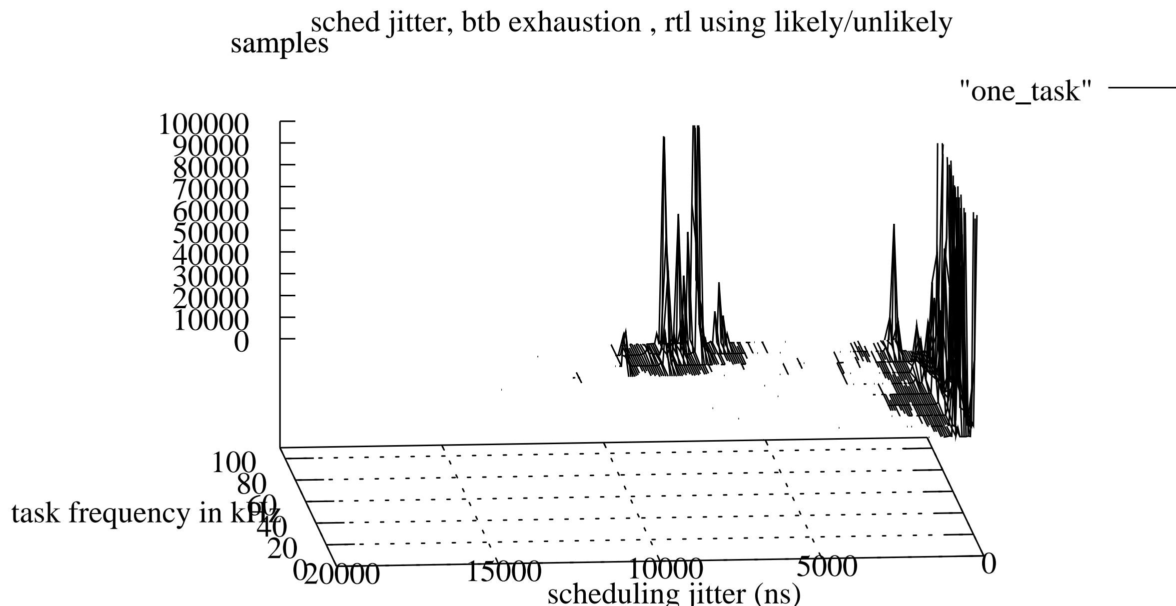

FIGURE 6: *caption*

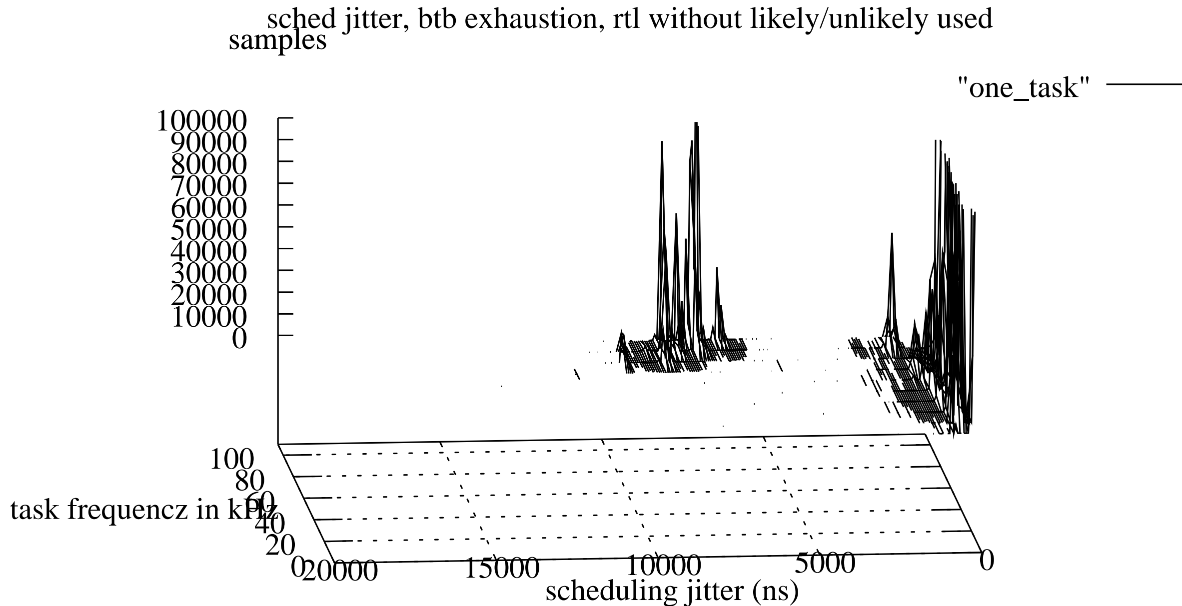

FIGURE 7: *caption*

4 Code optimization strategies derived from the above

The BTB and TLB related optimizations were already noted above, but beyond these some other strategies come into mind.

- inlineing of code
- explicit ordering
- conditional expressions
- local variables

4.0.5 Inlineing of code

Naturally inlineing of code (either explicitly with the __inline__ directive or via the -finline-functions gcc option to let gcc do it on its own) CAN improve performance - but unfortunately inlineing too much code will degrade performance - notably inlineing code that is rarely executed will trash intermediate cache lines and thus cause a high MM load that is counter productive (basically this is what GCCs likely/unlikely is all about - it tries to "inline" the body of conditional code properly).

For RT-systems the issue becomes more complex - in RT-systems rarely executed but time critical code may be better off in inline - though one must be aware of such measures being expensive with respect to impacting on non-RT (Linux) performance. As an example the interrupt interception generally is processing more non-RT interrupts than RT-interrupts - never the less inlineing the code for processing the RT-case will improve the RT-systems responsiveness.

When inlineing code there are three methods available (gcc-based tool-chain):

- -finline-functions to let the compiler do it
- in source:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

- header files:

```
__inline__
```

Our recommendation would be to let gcc do it at the first shot and then inspect the code generated as well as benchmark the performance before going into the lengthy process of manually inlineing. The rational behind this recommendation is two fold:

- manually inlineing can be painfully slow to do, and would need to be re-done on code changes - so if done at all, not before code really stabilized sufficiently !

- letting the compiler do it will most likely not yield the best possible output but can be done easily.

If GCCs inlineing shows significant changes of worst case performance then it most likely pays of to investigate in detail what can be inclined, and a manual cleanup could be considered.

4.0.6 explicit ordering

Locality of code has a big impact on the TLB - pulling all relevant code together will reduce the likelihood of TLB misses - TLBs are operating on pages - thus to improve tlb-miss-rates PAGE_SIZE aligned functions and function sequences are preferred. With respect to data items this means that traversing large data structures in PAGE_SIZE alignment will help - as a practical example, we reordered the RTLinux task-list to be ordered by priorities - thus the highest priority task can be located with a high probability of causing no D-TLB miss once the task-list it self was loaded , the impact of this explicit reordering is in the range of a few hundred nanoseconds to one microsecond on a 2GH AMD!

4.0.7 conditional expression

Simple rule - don't do:

```
 if(condition)
var = val;

        cmpl    $1, -4(%ebp) :condition
        je      .L3          :branch
.L2:
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
        ret
        .p2align 4,,15
.L3:
        movl    $2, -4(%ebp) :assignment
        jmp     .L2
```

use:

```
 var = codition ? var : val ;

        popl    %eax
        setne   %al
        andl    $255, %eax
        decl    %eax
        andl    $-33554, %eax
        popl    %edx
        addl    $33555, %eax
        pushl   %eax
        pushl   $.LC1
        movl    %eax, -4(%ebp)
        call    printf
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
```

instead - it will not require a jump instruction and also many platforms provide conditional load instructions (i.e. cmov on most X86). Unfortunately this rule might be strictly claimed for the simple case of conditionally setting a variable but for slightly more complex statements it is not generally predictable if the conditional (jmp based) execution path is less optimal than a conditional expression equivalent - in cases where this is in time critical code, inspection of the isolated assembler code should be done.

4.0.8 local variables

Not in all cases but in some - usage of local variables will improve system RT-performance, basically this is primarily a TLB/cache effect as the local variables are most likely to be within the same page address. So simply reducing the global variable stacked at the top of an application helps. We have not yet written and benchmark code to quantify the effect or to give a more explicit recommendation on scope issues (TODO). It also should be noted that due to gcc assigning local variables to registers before the coresponding assignment of global variables, local variables that end up as registers obviously have no cache/TLB side effects.

5 Conclusions

This section is notoriously incomplete - it will be updated along the path of finding and is focused on giving practically applicable information and not too theoretical "hints".

5.1 Data/Code locality

Analysis of user-space should focus on cache/tlb effects -¿ gprof,gcov and related tools.

Data locality should be a prime issue when designing and must be known for benchmarking as it is a critical parameter.

5.2 Interrupts

Interrupt scheduling is suboptimal in RTAI and in RTLinux (both GPL and Pro) as interrupts may interrupt interrupts - furthermore there is no interrupt priority available - thus application priority design into the interrupt service is not really possible.

The only way out at this point is signals. All currently available flavors of hard real-time Linux lack this capability and will need changes in this respect.