The offline scheduler for embedded vehicular systems

Raz Ben-Yehuda

BitBand Technologies Ltd.,
45 Hamelacha St., PO Box 8411,
Netanya, 42504, Israel
Email: razb@bitband.com

Yair Wiseman*

Computer Science Department,
Bar-Ilan University,
Ramat-Gan, 5290002, Israel
Email: wiseman@cs.biu.ac.il
*Corresponding author

Abstract: Nowadays, various transportation means use Linux as the operating system for their embedded control systems; however Linux uses all the processes in the hardware equally. This motivates some designer to develop an entire new operating system for the vehicle; where a modification of Linux can produce the same and even a better product. This paper explains how this modification was designed and implemented in a commercial transportation company.

Keywords: intelligent transportation systems; embedded vehicular systems; Linux; real time operating system; offline scheduling.

Reference to this paper should be made as follows: Ben-Yehuda, R. and Wiseman, Y. (2013) 'The offline scheduler for embedded vehicular systems', *Int. J. Vehicle Information and Communication Systems*, Vol. 3, No. 1, pp.44–57.

Biographical notes: Raz Ben-Yehuda is a Team Leader of embedded Linux real-time of robotics systems at Manz LTD, Petach Tiqwa, Israel. Before joining Manz, he worked as a Linux kernel developer for Bitband, ScaleMP and Monosphere.

Yair Wiseman got his PhD from Bar-Ilan University and completed two Post-Doc – one at the Hebrew University of Jerusalem and one in Georgia Institute of Technology. His research interests include avionics, vehicular systems, intelligent transportation systems, process scheduling, hardware-software co-design, memory management, real-time operating systems.

1 Introduction

Over the last years, Linux has been becoming the operating system of choice for many transportation systems. The Linux adoption incorporates a wide range of transportation means from small cars (Geer, 2004) to long trains (Jianhua, 2005; Carr, et al., 2006). Also, new embedded systems of aeroplanes employ Linux (Srovnal Jr and Kotzian, 2008) and some of them even a Linux distributed system (Kepner et al., 2004).

The adaptation of Linux has many advantages; however, Linux like other current operating systems has no easy way to assign a processor to do an individual service using an undisturbed, accurate and fast way and still being a part of the operating system address space as long as the processor is an active part of an operating system. We suggest a technique of dedicating one processor of a running machine to a specific task.

Such a feature is very useful for systems that propose means of transportation that use one computer to do more than one task like the vehicle proposed in the work of Xu et al. (2010) that is able to detect the fire and to navigate its way to it using the very same computer. Such a feature will be also beneficial in Carr et al.'s (2006) study where the authors suggest using the very same computer to accomplish two tasks – monitoring a train and making sure the driver never opens the doors on the side without a passenger platform.

Linux is also common in use for navigation systems (Walchko et al., 2003). In view of the fact that navigation systems are becoming a standard in many means of transportation, an embedded computer knowing to perform the navigation task besides another tasks can be beneficial.

Amdahl's law asserts that a linear speed-up is not likely to be practicable (Flynn, 1995; Eadline, 2007; Hill and Marty, 2008), so this can motivate in some cases sparing a processor for certain tasks. That is to say we can use the Linux kernel ability to virtually hot plug a (SMT) processor (Boutcher and Engebretsen, 2004); instead of letting the processor mark time in endless 'halts', assign the processor a service. The offline scheduler treats a processor as a device with computation ability and this is why the processor can be offloaded.

2 Related works

2.1 CPU sets

Current common technologies for service-oriented systems are Linux CPU sets (Derr, 2004; Shinde et al., 2008) and Solaris Resource Pool association (Solaris Dedicated CPU, http://docs.sun.com/app/docs/doc/8171592/gepsd?a=view; Gentzsch, 2001). Both of these technologies refer to assigning tasks to a set of processors, probably on the same memory node (in the NUMA case). CPU sets are actually a constraint on a (user space/kernel space) task to use resources available only in its set. CPU sets are used mainly in big systems, appliances, adapted to the client needs, where performance is an important issue. Assigning tasks to a CPU set is done using simple interfaces for maintaining memory policies and there is a very small overhead for the programmer. CPU sets was not designed to run real-time tasks and do not intent to reduce the interrupts over-head. This is unlike the offline scheduler who is a real-time scheduler. The offline scheduler may act as an alternate to softirqs and ISR. It is serialised, very accurate and provides a controlled environment to the service. The offline scheduler also protects the operating system in some cases. Therefore, Linux CPU sets and Solaris Resource Pool association serve different purposes than the offline scheduler.

2.2 INtime

INtime RTOS (tenasys, http://www.tenasys.com/products/intime.php) for Microsoft windows is an extension for Microsoft Windows running on any platform that utilises a standard Microsoft HAL: uniprocessor or multiprocessor. INtime RTOS has many components similar to the offline scheduler. INtime and the offline scheduler share this features:

- INtime is separated from Microsoft windows and offsched is separated from Linux; however both of the pairs share the same address space.
- Both INtime and offsched scheduler can allocate processor cores for exclusive use of INtime/offsched scheduler.
- Both INtime and offsched scheduler have high-precision system timer of 50 μs.

In spite of these, there are some other features that just one of the systems INtime or the offline scheduler maintains:

- INtime can identify user space tasks as real-time processes whereas the offline scheduler cannot.
- INtime has a real-time TCP-IP stack whereas the offline scheduler does not have.
- INTime can be used as a standalone RTOS whereas the offline scheduler cannot.
- INtime does not access Windows address space directly, whereas the offline scheduler accesses Linux address space directly.

The offline scheduler supports dynamic allocation of cores whereas INtime does not support.

2.3 IBM Logical partitions

IBM logical partitions (http://pic.dhe.ibm.com/infocenter/powersys/v3r1m5/index.jsp?topic=/iphat/iphatlparchoices.htm; Jann et al., 2003) divide a server's resources into several logical units. The bus is shared; the memory and the disc storage may be partitioned and some other resources like communications can be also part.

The partition model has a component similar to the offline scheduler; however as a concept the model is not relevant for the offline scheduler. In the offline scheduler system all the resources except the processor are shared. The offline scheduler can be depicted as a processor allocator; there is no administrator. Each processor has an access to some resources in the system, as long as it does not call 'schedule()'.

The component that is similar to the offline scheduler is a primary partition of IBM that may be depicted as CPU 0 and if an offloaded processor fails, the system will continue to run if the operating system memory is not corrupted.

2.4 RTOP

Any Linux machine may hang sometime. Hanging might happen due to a kernel crash or a system overload. What can be done when a server is remote? To detect a kernel crash 'netconsole' can be used; however what can be done when the system is overloaded? The server cannot be accessed because it is so loaded that SSHD (or any other daemon) does not get any CPU time. Utilities, vmstat, sar, top and so on, do provide details but under heavy load a user cannot access the machine remotely, or even locally. A user must be able to access the machine, and ask for the monitoring tool.

RTOP (Remote top) is monitoring software that provides top-like view of the system in any time. RTOP is based on the offline technology. This means that you must offline a processor (or enhance netconsole). RTOP is composed from RTOP utility ran in a remote Linux server and a driver that is invoked in the monitored server. RTOP driver pushes information out from the server. RTOP pulls information from the server.

3 The offline scheduler for vehicular operating systems

Vehicular systems are more often than non-real-time systems (Wiseman, 2010); nevertheless, there are also sometimes non-real-time tasks (Grinberg and Wiseman, 2007). Mckenney (2007) explains the relationship between the new trend of SMP machines and the boost of the use of Linux versions as a real-time operating system. The offline scheduler suggests a new version of Linux scheduler for multiprocessors real-time operating system. The new scheduler splits the processes into two groups – the processes of the traditional operating system and processes of the offline scheduler. Actually, the offline scheduler is a hybrid system, because the new system is both real-time and still a standard Linux server. The real-time property is mainly achieved by the NMI and the CPU isolation characteristics; however the offline scheduler still interacts with the operating system.

Caches typically maintain these properties: spatial locality, temporal locality and sequentiality. Keeping these properties in usual scheduling environments is not easy. In the offline scheduler platform, they can be easily maintained. The offline scheduler gives the programmer tools to control the cache miss rate. Low cache miss rate has a harmful effect on performance (Tomiyama and Yasuura, 1997; Drepper, 2007); hence the offline scheduler adapts some rules to handle the caching:

- Spatial locality: Accesses to same or near memory locations in the lifetime of a program are very likely. The offline scheduler abides this principle in its architecture. A user may assign a task to a processor without setting a different program into that cache. This way, the cache lines are not evacuated and the program data set and code are highly available.
- Temporal locality: Given a sequence of references to n memory locations, it is likely that following reference will be made into that sequence. The offline scheduler abides this rule same as spatial locality.
- Sequentiality: Given a reference to location n, it is likely that a reference to location $n + 1$ will be made in the near future. Processors utilise this rule by pre-fetching data.

Pre-fetching can be done in software or by the hardware. There is no true benefit in the offline scheduler but the fact that running NMI make sure no undesirable pre-fetches happen.

Traditionally, programs' size has been increasing (Flynn, 1995). Nowadays, programs' execution code size usually exceeds the L1i size. Most parts of a real-time program are characterised as non–real time whereas a relatively small portion of it is real time. When a real-time code shares cache storage with a management code, or IO code, it is flushed from the instruction cache (L1i) as well as its data (L1d and L2). The offline scheduler's architecture enforces a programmer to make the distinction between the real-time codes to the rest. This distinction maintains the real-time portion small.

If the working set is large enough, it is very likely that several memory locations will refer to the same cache line and cache flushes and misses may happen. The offline scheduler has no need to maintain memory pools like in the work of Reuven and Wiseman (2006). It has access to almost all the memory (exceptions are vmalloc'ed/ ioremap memory). A program having its own memory pool might cause congestion in cache lines if memory is not shuffled enough, but this will be a fault of the operating system; not of the offline scheduler.

Branch prediction is one of the processor's ways to speed-up code execution. Problem with branch prediction is that a branch might get flushed too early. Brach flushing happens mainly due to interrupts, exceptions and segments descriptor loads (System Programming Guide, 2002). The offline scheduler is running NMI so less pipeline flushes happen.

Multiple processors systems are usually designed for two reasons: faults tolerance and program speed-up. Current Linux versions have little respect to fault tolerance processors. Programs' speed-up is achieved by a program design and not enforced by the operating system. The offline scheduler is a system where all processors share the same memory. The offline scheduler architecture forces the programmer to write a program with a better speed-up because concurrent instances of the sub-units of the program run in each processor. Fault tolerance is partially achieved by the fact the even if the operating system stops (panics) or hangs the offline scheduler can still run and vice versa, if an offloaded processor failure does not corrupt the operating system's active memory the system will not stop. This is obviously cannot be compared to a failure in some user space task, so in this sense running a user space task is better. Multiprocessors systems are subjected to the same restrictions as a single-processor system, such as cache size, branch prediction, BUS speed, etc, which were discussed previously.

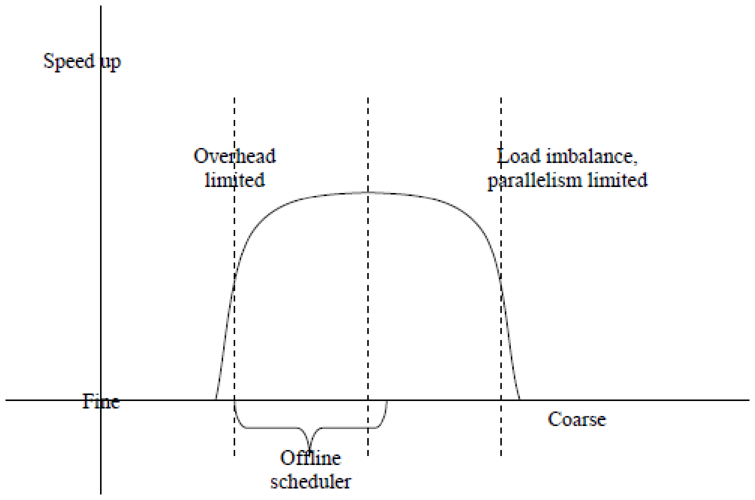
Partitioning is the process of dividing a task into sub-tasks, each of which can be assigned to a single processor. Partitioning a process is done in compile time. Partitioning objective is to reveal maximum amount of parallelism possible without exceeding machine's hardware limitation.

A partition analysis is preformed with some notion of a program overhead (Flynn, 1995). A program overhead (o) is the added time a task takes to be loaded into a processor prior to beginning execution. The larger the size of the minimum task, the smaller the partition overhead. Flynn (1995) describes program *P* work as follows: if a single-processor P1 program does operation O_1, the parallel version of P_1 does operations O_p where $O_p \geq O_1$. For each task T_i, there is an associated number of overhead operations o_i, so that T_i takes O_i operations without overhead then:

$$O_P^T = \sum^P \left(Oi + o_i\right) \geq O_i$$

where O_p is the total work that was done by P_p including overhead, and $o_i + 1 \geq o_i$. So, on one hand a better performance means maximising parallelism, and on the other hand finer code blocks increase the program overhead. Figure 1 depicts where the offline scheduler performs best.

Figure 1 Effects of grain size



At the offline scheduler, $o_i \rightarrow 0$, so there is no program overhead. This is because there is no stack manipulation, and all tasks are in the processor's cache. Load may be imbalanced only if several processors handle different resources and tasks. For example, if one processor handles network interface A, and second processor handles network interface B, and the load on the two interfaces is not the same (Wiseman et al., 2004). The offline scheduler cares less for load imbalance as long as the service is being done. If not, there is something wrong with the design. When load is well-balanced, we can say that:

$$O_i \approx O_1 \Rightarrow O_p^T = \sum^p i(O_i + 0) = O_{1p}$$

Sometimes two tasks depend one on the other. In a single processor a typical scenario is:

1 T1 put message on T2 queue.

2 T1 schedule

3 Do something

4 T2 gets processor time

5 T2 replies on for T1

6 T2 schedules

7 Do something

8 T1 gets processor time

9 T1 gets T2's reply

The offline scheduler behaves very much the same. Yet, when using the offline scheduler, T1 may decide to schedule T2 directly, and only when T1 lets T2 will get a processor time, i.e. T1 chooses when T2 get processor time. Phases 7 and 3 do not exist in the offline scheduler. This small difference is extremely important when hard real time is required.

Scheduling can be performed either dynamically or statically. Static scheduling is determined during compilation and dynamic is performed in real time. Ngai (1992) observes that all major issues in run-time scheduling focus in insuring performance and reduce overhead losses. In multiple processors this is even a more important issue (Wiseman and Feitelson, 2003). The following four run-time scheduling major overheads include:

- Gathering information: Gather system and programs real-time information. This includes, control structure, identification of critical paths and dependencies. Dynamic information includes work loads and resource availability. The offline scheduler has no need of gathering information. It is up to the designer to decide what information will be used, e.g. the offline scheduler timer uses 'rdtsc' counters and Rdtsc has a very small overhead.
- Scheduling: In the usual course of events, scheduling is performed as shown in Table 1.

When using the offline scheduler, scheduling is performed in run time and has a very little overhead. The offline scheduler has a single-stack context (Wiseman et al., 2008). The offline scheduler scheduling is mere procedure invocation. There are two scheduling procedure in the offline scheduler:

- offsched_schedule: A routine is a registered in the offline scheduler pending queue. In the offline scheduler there is both fast scheduling, simplicity and hardly any overhead.
- The offsched reschedule: A routine for assignment of a task into offsched run queue (calendar). It is lockless and very simple.
- Dynamic execution control: This includes clustering and process creation at run time. In the offline scheduler, a process creation is the actual assignment to a designated processor (offsched schedule); there is no creation of a context as in the familiar UNIX. There are no pids, gids or similar features. There is no notion of clustering of processes.
- Dynamic data management: This includes a minimisation of memory overhead delay when accessing data. Delay may improve by assigning tasks by a policy of a minimum memory delay. In the offline scheduler, memory access is matter good scheduling design. Due to the offline scheduler serialised nature, memory low latency scheduling policy can be easily achieved.

Table 1 Compile time vs. run time in scheduling

	Advantages	*Disadvantages*
Compile Time	Less overhead	Lack of fault tolerance Compiler lacks stall information
Run time	More efficient execution	Higher overhead

Consider the following scenario: process A assigns a message to process B. Process A schedules process B after itself and exit. Process B runs right after and the memory is in the processor cache. When process B is completed, it will schedule process A and will exit. During this process, there is no cache miss as long as the data working set is smaller than the processor's cache.

4 Communication and synchronisation

A typical programmer does not program when the processor cache or MESI (Illinois Protocol) in his mind. He/she does not know even on what hardware his/her program will be running on. So, his/her program is likely to have in-efficient memory accesses. If the program is intended to be running on multiprocessor machine, queues and synchronisation primitives will be used. So cache misses, locks (atomic increment must seize the bus in a, SMP system) and pre-emption are likely to load the system. The offline scheduler serialised nature reduces this contention.

One may ask, why not create a busy loop like:

```
While (can I do my thing)
do my thing
```

This code will create an RCU starvation. RCU is a synchronisation technique which enables multiple readers and multiple writers to access mostly accessed data structures from multiple processors. RCU starvation happens because each processor must walk through a quiescent state (Bovet and Cesati, 2006). A quiescent state is when one of the bellow happens:

1 A processor performs a context switch.

2 A processor executes user mode.

3 A processor executes the idle loop.

And neither of these will happen in a busy loop. The offline scheduler eliminates RCU starvation since the processor does not have to walk through a quiescent state as it is not part of the operating system.

Consider the following sequence of operations:

```
INC RAX
ADD RCX, RBX
```

These instructions are independent. In regular pipelined processors these two instructions are not likely to run concurrently. Some commands might be associated with interrupts and exceptions. If the interrupt service routine runs in the initiating processor context, RCX and RBX content will be lost unless some additional logic is added to the processor (Harry, 1997). Either case, a processor has to do additional work that reduces performance. The NMI property of the offline scheduler actually makes programs running in the offline scheduler context faster than in an active processor. When designing multithreaded software, one has to understand the price of a context switch with respect to the system requirements.

Moore law presents us a simple statement, if in 2008 we have 4 cores in a single die, in 2010 we will be having 8 cores, in 2012 it will be 16 cores, in 2014 it will be 32 cores, etc. Does the current Linux operating system design fits the multicore era? Should the operating system handle so many cores? As this paper argues, the answer is: 'not always'. Having an operating system balancing tasks between 16 processors is not necessarily good. The Linux kernel migrate pages and move tasks between processors repeatedly to achieve balance in the system. The offline scheduler on other hand does not care much for imbalance; it aims for responsiveness, accuracy and simplicity. The offline scheduler can produce linear speed-up as long as it is not bounded by BUS speed.

5 The offlet

In the offline scheduler environment, we do not use the terms threads, processes or softirqs. The offline scheduler refers to a processor as a device running an offlet. Offlet is the context of an offlined processor. When an offlined processor is removed from the operating system, it is simply set to the 'halt' machine instruction while interrupts are disabled. Right before the halt instruction is executed, the offline scheduler is invoked. From now on, we are in an offlet context. Offlet context has some restriction that a user must be aware of:

1 A user may not cause any page fault, meaning he cannot access vmalloc'ed memory (only by 'walking on the pages').

2 A user may not 'STI'.

3 A user may not invoke any code path that ends up in 'schedule'.

This is why the author refers to the offline scheduler context as an offlet; it is because it has more constraints than any other kernel context. Offlets may be scheduled in time T, when T can be any time starting from now. Offlets may be set up on the stack, schedule and released; very much like any other kernel entity but much faster.

The offline scheduler is based on the notion that processors are not an expensive resource. In general, when task A is in kernel context and wishes to seize resource X all it does:

Step 0:..

Step 1: Lock

Step 2: ..

Step 3: Unlock

Step 4:..

If Step 4 is not serialised with respect to the previous steps then offlet will suggest an asynchronous service, instead of spinning in Step 1, the offline processor is asked to do Steps 1–3 for the user. Offlet scheduling is very light so the scheduling cost is negligible in terms of computation. Another added value of offlets is that a system may choose to serialise access to a device through an offlet. A good example is offline NAPI, which is described below. This serialisation actually relieves the kernel from contention on slow devices.

When any context enters the vmalloc area, it must generate a page fault in each processor this if the referenced address was not referenced before on the same processor. This is because vmalloc area is a dynamic-mapped area and the processor's MMU has no reference to it. Kmalloc address space is set on the kernel static pages that never change. For an offlet to access vmalloc area, or user pages, it must walk on the pages. Walking on the pages means dismantling the address to pages and calling kmap atomic on each page.

6 The offline timer

The offline scheduler timer is a plain busy-pause loop running at the frequency of the CPU (in our tests is has been 2 GHz). Each time unit is divided into N slots, e.g. 1 ms is divided into slots of 1000 μs. There is still a need to adjust the timer, because it uses the rdtsc mechanism which might be inaccurate. If a 100 μs resolution meets the requirements, HPET overhead will be quite small, about just 3% CPU usage in our tests (Lenovo T61 laptop). So, in such a case the offline scheduler timer will be redundant.

In cases when just few timer handlings are needed, i.e. few timer handlings every several hundreds milliseconds, one can use TICKLESS (kerneltrap, http://kerneltrap.org) timer implementation.

However, since an operating system does other tasks, like processing packets, an interrupt might be nested and thus be behind schedule. In addition, when the amount of timers is larger, the accuracy is less important.

7 Offline NAPI

NAPI (Bolla and Bruschi, 2005) stands for New API. NAPI is a technique for interrupts mitigation in networking in the Linux kernel. NAPI technique is a simple polling over incoming packet arrival queue; this eliminates the constant interrupt processing. By default, Linux uses receive interrupt for packet processing, and activate NAPI when a certain threshold of number of incoming packets is reached. This threshold is known as the network 'weight'. Only one processor may call poll, this is because only a single processor can get the initial interrupt. There are some flaws of NAPI.

1 Latency: In some cases, NAPI may introduce additional software IRQ latency.

2 IRQ masking: On some devices, changing the IRQ mask may be a slow operation, or requires additional locking. This overhead may negate any performance benefits observed with NAPI.

3 Rotting Packet: In some cases a packet may rot in the RX ring of the device. This happens due to a race between the time we decide we have no packet in the RX ring and the time we are forced to enable interrupts.

The offline scheduler NAPI is aimed to solve the above problems and provide infrastructure to other services for packet processing.

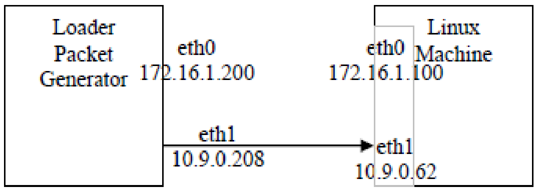
- Latency: Very much like NAPI, the offline scheduler NAPI will process incoming packets from RX ring. Unlike NAPI, spinning processors will have very little latency (if any) so disabling RX interrupts entirely is possible, this way we eliminate both problems. In addition, since we have no initial interrupt (an interrupt that triggers NAPI) we can have several processors spinning over the same device.
- IRQ masking: The duration of the operation of IRQ masking is not a problem because it is done only once. Once IRQ masking is complete, the offline scheduler NAPI is not bothered with it again.

- Rotting Packet: Because interrupts are never enabled and the operating system continuously spins over the network devices, this problem does no longer exist.
- SMP IRQ affinity: Current IRQ affinity is in a device granularity. In offline NAPI, granularity may be based on service affinity. Meaning, instead of having all packets of a device routed to a group of processors, packets of type A may be assigned to processor X, type B to processor Y and so on. The rational here is that there is no sense in putting together ARP queries on the same processor as the user's application's processor.

8 Evaluation

We have generated traffic through eth1 to eth1. Each machine has been connected to two different network segments. Traffic has been generated on the 10.9 segment. Figure 2 depicts this test. The grey area is where the offline firewall process runs. It controls all interfaces. The test was conducted on a Supermicro PDMSI machine, with an Intel Pentium 3.4 GHz.

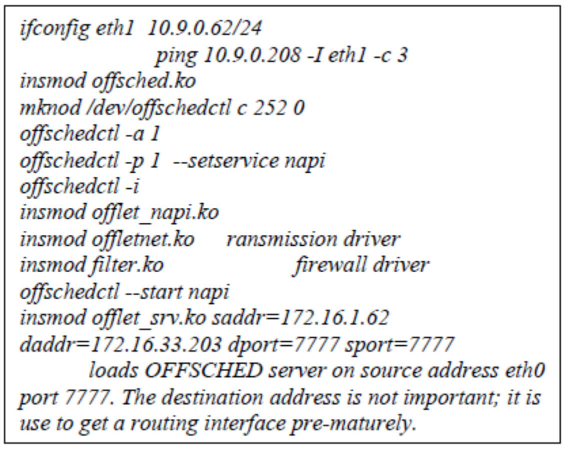
Figure 2 Traffic configuration



Hyper threading enabled. Receiving interface was Intel 1Gbps 82546EB. e1000 and Linux kernel version was 2.6.30. The Loader is generating traffic Linux packet generator driver, also is known as pktgen. Pktgen UDP port is 9, which is the discard port.

Figure 3 details the loading procedure. The test was first conducted with Linux Native NAPI. Then CPU1 was dropped and offsched was run.

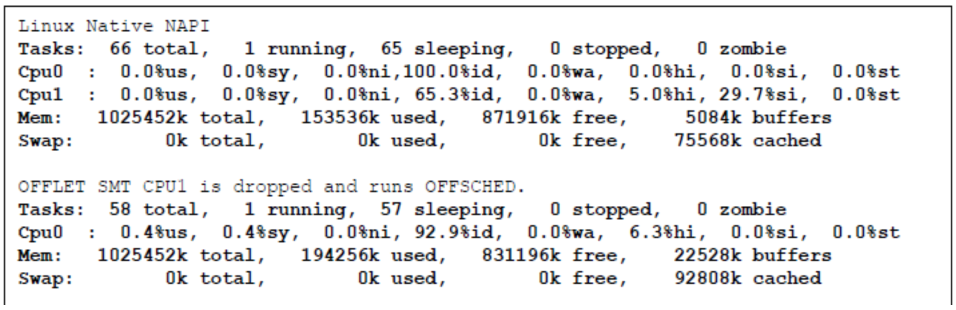
Figure 3 Loading procedure



```
ifconfig eth1  10.9.0.62/24
                   ping 10.9.0.208 -I eth1 -c 3
insmod offsched.ko
mknod /dev/offschedctl c 252 0
offschedctl -a 1
offschedctl -p 1  --setservice napi
offschedctl -i
insmod offlet_napi.ko
insmod offletnet.ko      ransmission driver
insmod filter.ko                  firewall driver
offschedctl --start napi
insmod offlet_srv.ko saddr=172.16.1.62
daddr=172.16.33.203 dport=7777 sport=7777
          loads OFFSCHED server on source address eth0
port 7777. The destination address is not important; it is
use to get a routing interface pre-maturely.
```

Figure 4 shows the result of native NAPI versus the offline scheduler. Using the offline scheduler, the hardware interrupts was 6% because receiving interrupts has not been disabled. When using Native NAPI, the consumption of CPU1 was 35%, whereas cpu0 is mostly idle. In the offline kernel CPU0 is 7% busy due to hardware interrupts.

Figure 4 Native NAPI versus offlsched



```
Linux Native NAPI
Tasks:  66 total,   1 running,  65 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.0%us,  0.0%sy,  0.0%ni, 65.3%id,  0.0%wa,  5.0%hi, 29.7%si,  0.0%st
Mem:   1025452k total,   153536k used,   871916k free,     5084k buffers
Swap:         0k total,        0k used,        0k free,    75568k cached

OFFLET SMT CPU1 is dropped and runs OFFSCHED.
Tasks:  58 total,   1 running,  57 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.4%us,  0.4%sy,  0.0%ni, 92.9%id,  0.0%wa,  6.3%hi,  0.0%si,  0.0%st
Mem:   1025452k total,   194256k used,   831196k free,    22528k buffers
Swap:         0k total,        0k used,        0k free,    92808k cached
```

The Intel network interface was loaded with 1 Gbps of incoming traffic as observed by bmon. In terms of pure processor consumption, 107/200 is the total processing consumption of offline NAPI while in native NAPI it is 35/200. So, one may wonder, are not we wasting a processor? The answer is actually yes, we do. But what we do earn is an operating system running undisturbed. If this machine has been loaded with more traffic, then the operating system may be too loaded to even be accessed. In a machine employing offline scheduler, this inaccessible state will not occur, because a denied packet processing is confined to the offloaded processor.

9 Conclusion

Linux is a good platform for embedded systems and particularly for vehicular autonomous systems (Williams and Bergmann, 2004; Yaghmour et al., 2008). Adding the offline scheduler will enhance the Linux abilities. The offline scheduler joins two different system types in a single machine, a real-time system and a standard Linux machine. This is very helpful for systems as found in the Carr et al.'s (2006) study, where the authors use the embedded computer to monitor the train speed (a real-time task) and to make sure the driver never opens the doors on the side without a passenger platform (a non-real-time task). Offloading processors to serve as dedicated engines might be very helpful in various real-time scenarios. Using the offline scheduler, there is no need to buy expensive offloading network cards when the same feature can be achieved by commodity hardware. Who does need network processors for this feature, when every standard processor can do the same thing?

References

Bolla, R. and Bruschi, R. (2005) 'High-end Linux based open router for IP QoS networks: tuning and performance analysis with internal (profiling) and external measurement tools of the packet forwarding capabilities', *Proceedings of the 3rd International Workshop on Internet Performance, Simulation, Monitoring and Measurements (IPS MoMe 2005)*, pp.203–214.

Boutcher, D. and Engebretsen, D. (2004) 'Linux virtualization on IBM POWER5' systems', *Ottawa Linux Symposium*, Ottawa, Ontario, pp.113–120.

Bovet, D.P. and Cesati, M. (2006) *Understanding the Linux Kernel*, O'Reilly Media Inc., Sebastopol, CA.

Carr, D.W., Ruelas, R., Salcedo-Becerra, H. and Ponce-Castaneda, G.A. (2006) 'A Linux-based system to monitor train speed and doors for a light-rail system', *8ht Real-Time Linux Workshop*, Lanzhou, Gansu, China.

Derr, S. (2004) *CPUSETS*. Available online at: http://lxr.linux.no/linux+v2.6.26.3/Documentation/cpusets.txt

Drepper, U. (2007) *What Every Programmer should know about Memory*. Available online at: http://people.redhat.com/drepper/cpumemory.pdf

Eadline, D. (2007) 'Multi-core melee', *Linux Magazine*, No. 80, pp.40–41.

Flynn, M.J (1995) *Computer Architecture Pipelined and Parallel Processor Design*, Jones and Bartlett Publishers Inc., Boston, MA,

Geer, D. (2004) 'Survey: embedded Linux ahead of the pack', *IEEE Distributed Systems Online*, Vol. 5, No. 10, 3p.

Gentzsch, W. (2001) 'Sun grid engine: towards creating a compute power grid', *1st IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, Brisbane, Qld., pp.35–36.

Grinberg, I. and Wiseman, Y. (2007) 'Scalable parallel collision detection simulation', *Proceedings of Signal and Image Processing (SIP-2007)*, Honolulu, Hawaii, pp.380–385.

Harry, D. (1997) *Computer Organization for Multiple and Out-of-Order Execution of Condition Code Testing and Setting Instructions*, US Patent 5630157.

Hill, M. and Marty, M. (2008) 'Amdahl's law in the multicore Era', *IEEE Computer*, Vol. 41, No. 7, pp.33–38.

Jann, J., Browning, L.M. and Burugula, R.S. (2003) 'Dynamic reconfiguration: basic building blocks for autonomic computing on IBM pSeries servers', *IBM Systems Journal*, Vol. 42, No. 1, pp.29–37.

Jianhua, Z. (2005) 'Design of electrical monitoring and control terminals for trains based on Linux', *Industrial Control Computer*, Vol. 4, No. 1, pp.3–5.

Kepner, J., Moore, M., Travinin, N., Kim, H., Reuther, A., Currie, T., McCabe, A., Mathew, B., Rabinkin, D. and Rhoades, A. (2004) *Deployment of SAR and GMTI Signal Processing on a Boeing 707 Aircraft using pMatlab and a Bladed Linux Cluster*, Technical Report, Massachusetts Institute of Technology, Lexington Lincoln Lab.

Mckenney, P.E. (2007, January) 'SMP and embedded real-time', *Linux Journal*, No. 153, p.1.

Ngai, T.F. (1992) *Run Time Resource Management in Concurrent Systems*, PhD Thesis, Department of Electric engineering, Stanford University.

Reuven, M. and Wiseman, Y. (2006) 'Medium-term scheduler as a solution for the thrashing effect', *The Computer Journal*, Vol. 49, No. 3, pp.297–309.

Shinde, P., Sharma, P. and Guntupalli, S. (2008) 'Automated process classification framework using SELinux security context', *3rd International Conference on Availability, Reliability and Security (ARES 08*), pp.592–596.

Srovnal Jr, V. and Kotzian, J. (2008) 'Development of a flight control system for an ultralight airplane, international multiconference on computer science and information technology', *IMCSIT*, pp.745–750.

System Programming Guide (2002) *IA-32 Intel ®Architecture Software Developer Manual*. Available online at: http://pdos.csail.mit.edu/6.097/readings/intelv3.pdf

Tomiyama, H. and Yasuura, H. (1997) 'Code placement techniques for cache miss rate reduction', *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, Vol. 2, No. 4, pp.410–429.

Walchko, K.J., Nechyba, M.C., Schwartz, E. and Arroyo, A. (2003) 'Embedded low cost inertial navigation system', *Florida Conference on Recent Advances in Robotics*, FAU, Dania Beach, FL.

Williams, J. and Bergmann, N. (2004) 'Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip', *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, 21–24 June, pp.163–169.

Wiseman, Y. (2010) 'Take a picture of your tire', *Proceedings of IEEE Conference on Vehicular Electronics and Safety (IEEE ICVES-2010)*, Qingdao, ShanDong, China, pp.151–156.

Wiseman, Y. and Feitelson, D.G. (2003) 'Paired gang scheduling', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 6, pp.581–592.

Wiseman, Y., Isaacson, J. and Lubovsky, E. (2008) 'Eliminating the threat of kernel stack overflows', *Proceedings of IEEE International Conference on Information Reuse and Integration (IRI 2008)*, pp.116–121, Las Vegas, USA.

Wiseman, Y., Schwan, K. and Widener, P. (2004) 'Efficient end to end data exchange using configurable Compression', *Proceedings of the 24th IEEE Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, pp.228–235.

Xu, Y., Li, D., Chen, S. and Chang, H. (2010) 'Intelligent fire-fighting vehicle model based on embedded system', *Journal of Computer Applications*, Vol. 30, No. 2, pp.560–563.

Yaghmour, K., Masters, J., Gerum, P. and Ben-Yossef, G. (2008) *Building Embedded Linux systems*, O'Reilly Media Inc., Sebastopol, CA.