

Performance Considerations for the use of Linux in Embedded Systems

7. Juni 2004

Legal Notice

Dieser Beitrag ist lizenziert unter der GNU Free Documentation License.

Zusammenfassung

The use of Linux for embedded devices is continually growing. Embedded devices by their very nature are often limited in terms of their computational power as well as their available memory. However these devices are typically targeted for specialized applications and are frequently equipped with special hardware that allows the acceleration of some of the most important operations.

This talk will cover the performance considerations going into the design and implementation of an embedded device discussing the architectural decisions, hardware design and redesign, porting challenges, device driver adaptation and various changes that might be necessary to the Linux kernel in order to obtain optimal performance.

Of special interest are the design and implementation of specialized network and block device drivers. The general characteristics of the device interfaces of Linux poses certain challenges in the development of device drivers for embedded systems. The focus is mostly on Linux 2.4.X but recent developments of the 2.6.x Linux Kernels are also considered.

Version 1.0

Performance Considerations for the use of Linux in Embedded Systems

Christoph Lameter, Ph.D. (christoph@lameter.com <<mailto:christoph@lameter.com>>) Professor for Information Technology and Philosophy University of Phoenix (<http://www.phoenix.edu> <<http://www.phoenix.edu>>)

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

1 Embedded Devices - An Introduction

What are embedded devices? The short answer is that an embedded device is a combination of hardware and software designed to perform a specific function. The device is typically dedicated to one purpose alone and in contrast to a PC, which is designed to be a general purpose computer is not designed to be able to run general applications that a user might install.¹

Examples for embedded systems are routers, cell phones, digital watches, DVD players, DVRs, engine controllers, Missile Guidance Systems, GPS receivers, calculators and microwave ovens. Embedded devices can be found in many of the things we use for everyday activities and their number is increasing rapidly. Embedded devices are an important way to make the things we use more intelligent and more useful to us. General purpose computers (*Personal Computers*) are composed of numerous embedded systems like the keyboard controller, hard disk controllers, CRT controller and so on. Each of those may have their own processor and logic.

¹Jack G. Ganssle and Michael Barr, *Embedded Systems Dictionary*. CMP Books, 2003, s. v. "embedded system. <<http://www.netrino.com/Publications/Glossary/>> >.

The first embedded system known was the guidance system for the Apollo Moon project ², which was the precursor to the 4004 CPU, which in turn was followed by the 8008, the 8080, the 80186, 80286, 80386, 80486 and the Pentium. If you are an professional in the Information Technogy then you surely know how the sequence continues up to today. It is noteworthy that the microprocessor began as an embedded device and not as a general purpose computer.

The first section develops a categorization of the kinds of embedded devices that exist and are of interest to the considerations here. Then follows a general discussion of the use of Linux on these devices. Performance considerations for embedded devices are discussed in general and then several particular issues are taken on. These are first the effects created by the necessity to manage memory in small 4KB chunks and the problems often encountered with maintaining cache coherency in but also the performance considerations going into the design and implementation of network and block device drivers.

User space may also pose an important challenge given the complexity of todays Linux applications and therefore a short section on that topic follows discussing a variety of approaches taken to decrease the complexity of Linux applications and tools.

An alternative frequently used instead of Linux are real time operating systems. There are enhancements for Linux available that make real time functionality available in Linux. These are discussed in the final section before the conclusion.

2 Types of Embedded Devices

Embedded devices are basically computers designed for special purposes. They can roughly be categorized according to their complexity and by how far away they get from the general purpose character of the PCs. In each category discussed in the following paragraphs we encounter less CPU power, less memory and power consumption:

2.1 MiniPC or the Living Room PC

These systems have typically the full functionality that is also offered by a regular PC. However, if they have a video output this will typically be channelled through a video feed suitable for a TV and not to a monitor. Instead of a keyboard a remote control is in use. These devices may still have connectors for a keyboard and monitor. Since these devices are typically used in the living room, they have to be silent and therefore typically have no fan. This means that the CPU is restricted by heat that can be handled by a heatsink. At this time the VIA EPIA boards with C3 processors are popular for these devices since they only generate minimal heat and are still working at speeds of around 800Mhz. These EPIA boards are typically used to realize custom media centers. They are mostly capable of running regular Linux distributions but require special device drivers for their multimedia features. These systems are just limited in their resources but keyboard, monitor and mouse are attachable and the device can then also be used like a general purpose PC. The processing power of these device is typically about a fourth of that of a regular PC.

2.2 MicroPC (uPC)

One the low end of the MiniPC class of embedded devices one ends with CPU speeds from 100-240Mhz, around 16-128M of memory and a few hundred megabytes or a few gigabytes of storage. This is at the low end where the common Linux distributions are still useful. As the CPU power becomes less, more and more special software builds become necessary. The processing power is less than a tenth of regular desktop PCs. Regular PC keyboards and monitors cannot be connected to these devices but they might offer command line access via a serial interface (typically in use during the development or debug phase for these devices) or be reachable via a network interface. Special measures are necessary if these device must process multimedia streams. Examples of devices in these categories are the game consoles (GameCube, Xbox and their friends) and common consumer grade home gateways and routers. Specialized Linux versions are frequently used in these devices.

²George Tomayko "Computers on Board the Apollo Spacecraft in *Computers in Spaceflight: The NASA Experience*. NASA, 1987. <http://www.hq.nasa.gov/office/pao/History/computers/contents.html>

2.3 Componentized minimal systems (CMS)

CMS are still composed of a few separate chips but these chips have highly integrated functionality. The CPU frequencies drop to 20-150Mhz and I/O support becomes very limited. Examples of these types of devices are Cell phones, PDAs and many other miniaturized devices that can be carried in a pocket. Recently gumstix has brought out a system of minimal sizes that can still be programmed using highly customized Linux distributions but in general these devices have firmware that cannot be changed. The upgradability of these systems and the ability to run other software is very limited. At this point it is also usual to sacrifice the MMU (Memory Management Unit) and therefore special versions of Linux (like uCLinux) may have to be used with these devices.

2.4 Microcontrollers or a System on a Chip (SOC)

Microcontrollers typically have very specialized programming modes and these are therefore frequently not usable with Linux. However, the high degree of integration seems to be leading to the development of complete PCs on single chips. For example in 2002 ZF Microsolutions developed the ZFx86 (but then became tied up in legal proceedings). Linux Devices has a listing of existing single chip solutions that might be used in place of the older micro controllers.³ However, while one should note the intentions to develop single chip systems, it needs to be realized that most of these systems advertised as SOC still require multiple components for a functional system. These are more like the componentized minimal systems mentioned in the last section.

3 Linux on Embedded Devices

In the area of software for embedded devices, Linux competes with a variety of other operating systems. The reasons for the use of Linux in the embedded systems are the following advantages:

1. *No licensing fees.* Linux is readily available and does not require a license agreement. This implies flexibility and no royalties for the operating system of the devices using Linux.
2. *Customizable.* All of the Linux source code is available and is modifiable without restrictions. It is therefore possible to customize an operating system kernel for a specific embedded device without having to negotiate access to the source code with a vendor.
3. *Support for a large number of platforms and devices .* Linux has the broadest coverage in terms of platforms supported of all operating systems. If a popular CPU is used then it is highly likely that a Linux Port to that CPU already exists. If such support does not exist, then it is typically easy to port Linux to the new platform. There is a large selection of hardware drivers already available which may be directly useful for the embedded device or which may be customized for the device.
4. *Uniformity of Software from smallest devices up to the largest systems .* Software running under Linux is portable. Software can therefore be developed on desktop machines and then be recompiled for an embedded device. Typically this allows a much faster development cycle. Many software components available for server and desktop Linux systems can be reused for embedded systems.

4 Performance Considerations

Performance is not the major consideration in the design of an embedded system. There are usually other requirements like size, costs of components and power consumption that are primary factors when developing an architecture for an embedded device. CPUs used for general purpose workstations today have significant power requirements and generate large amounts of heat. The requirement for low power consumption, non availability of fans and limited space lead to the necessity of having CPUs with relatively low speeds. The problematic issues arising from the platform are frequently only realized when the software is expected to

³<http://www.linuxdevices.com> <<http://www.linuxdevices.com/>> . Search for "SOC or "System on Chip. See Rick Lehrbaum, *One-chip Linux systems hasten arrival of Post-PC Era* (LinuxDevices.com, 2000).

get the maximum performance out of the hardware that is already nearing production. This provides unique challenges because Linux was mainly designed for a general purpose PC where performance is one of the main design criteria.

Embedded devices may have comparably slow processor and I/O capabilities but these devices are frequently equipped with special hardware features that accelerate critical tasks of the device. For example an embedded device that processes media streams might have accelerators that allow moving data in memory with a much higher speed than the low-powered CPU could do on its own. The challenge to using Linux on embedded devices often results in the need to integrate special hardware into the system software to reach the expected performance.

5 The 4KB page size issue

Linux organizes its memory in 4KB pages and uses a MMU (Memory Management Unit), that is either build into the CPU or external, to realize paging by reading and writing 4KB chunks of memory. A file of 1 Megabyte in length might be scattered over 256,000 4KB blocks located at seemingly random locations in memory. Writing or reading large files from any I/O device must serialize these randomly scattered blocks in order to perform read or write operations. A desktop CPU may have sufficient computational power to handle each of these pages separately. However, in embedded systems the significantly lower CPU speed poses a severe load on the CPU. Large scale I/O realized by software will therefore result in extremely slow transfer rates on embedded devices.

It is therefore often necessary that an embedded device be equipped with special accelerators to perform these transfers. There will still be a significant computational impact since the CPU has to organize lists of these 4KB page locations that are then passed to the hardware. The meta data of the pages still has to be managed but the major work of transferring lists of pages to I/O device can be delegated to the hardware. Having to continue the transfer at different locations every 4 kilobytes may still cause performance issues since hardware these days typically fetches data from memory in bursts and is optimized to deliver optimum memory throughput only when data is read in sequence. It is in particular difficult if the hardware accelerators can process arbitrary amounts of memory but must be reprogrammed for each transfer separately. This would mean that these devices must be reprogrammed for each transfer of 4KB (often less!) separately which makes the hardware accelerator ineffective.

The 4KB page size is in particular a problem for Linux 2.4.X kernels which randomly scatter the 4KB blocks throughout memory. In Linux 2.6.X changes were made to the way these 4KB blocks are located in memory. 2.6.X kernels organize pages that are accessed in sequence also sequentially in memory. This is then used to coalesce the 4KB blocks into larger segments improving the speed of I/O by reducing the number of necessary descriptors for a transfer and increasing the sizes of the blocks passed to the hardware acceleration units for I/O. As a consequence I/O efficiency typically rises significantly just by switching from 2.4 Linux kernels to 2.6 Kernels. Hardware accelerators become much more effective.

The 4KB page size is recognized to be a significant issue for large memory transfers since the management of the meta data for each 4KB page causes significant overhead. The high performance Linux systems by SGI for example can increase the page size to 64K to address this issue. A discussion in 2003 and 2004 on the Linux Kernel mailing list shows general support for increasing the default page size significantly for the 2.7 developmental kernels to improve overall system performance.

Kernels can be build for a larger page size today. However, it is difficult to change the default page size in 2.4 and 2.6 kernels since many tools and many drivers depend on the page size being 4KB. Hardware (such as the Pentium CPU) typically has special support for a 4KB page size. The file format for binaries locates code at 4KB boundaries because executable files are used for paging. A change of the page size would require the rebuilding of all binaries and might even require a reworking of the binary format used on a platform. The proposed change of the page size in 2.7 will likely break the binary format for executables and it is therefore likely that the next stable release of the Linux Kernel will no longer be a 2.X kernel but 3.0. However, the increase of the page size seems to be the best solution to address the overhead generated by the page meta data and increase the efficiency of an embedded device.

6 Cache coherency and Cache flushing

Some of the simplest embedded devices have no CPU cache at all and therefore this section might not apply to all embedded devices. However, given the speed issues accessing DRAM today, a cache is inevitable if speeds over 100Mhz of processing speed are expected. The effective handling of the data and instruction cache is important to be able to utilize the CPU at optimum speed.

Embedded devices may have problems with cache consistency. Cache inconsistencies arise if pieces of memory are moved to the cache. The cache is considered to be authoritative on what is the current state of in memory by the CPU. However, an I/O device might be able to perform direct I/O bypassing the CPU and the CPU cache. If the CPU now accesses a data location it will get the value from the cache which is not reflecting the true state of the DRAM cell. These inconsistencies typically result in data structures being corrupted.

Maintaining cache consistency can be achieved in a variety of ways. General purpose CPU's support *bus snooping*⁴ realized through one or the other cache coherency protocol implemented on the memory bus. This generally means that the CPU has to listen to the bus even after bus control has been given to another piece of hardware for DMA. The CPU will listen to the bus and invalidate cache cells if their content is being written by the DMA transfer or may provide its cache contents if a memory cell is requested that has not been flushed to memory yet.

However, bus snooping means that the bus interface of the CPU must stay active even if the CPU is idle. This requires significant power resources since the bus interface must be powered even if the CPU not using the bus. It also increases the complexity of the bus interface and the cache to a significant degree since cache coherency protocols are complex, require advanced timing sensitive new bus behavior and become more complex for more advanced hardware. Moreover cache coherency protocols do not scale well and might reduce the speed increase obtained by the use of a cache.

The simple solution frequently realized in embedded devices is to have *software control* of the cache. When writing a device driver the programmer has to insert control statements to flush the cache at appropriate moments to guarantee the consistency of the data in memory before a transfer or to insure that nothing is cached in an area where a DMA device is expected to transfer data to.

The advantage of software control of the cache is a much simpler design of the bus interface and the ability for devices to go off the bus thereby saving precious power. The disadvantage is that software control shifts the burden of maintaining cache coherency away from the hardware to the software developer. The software developer is a human and therefore coding errors may become a significant cause of cache coherency problems.

Frequently CPUs support special software instructions to control the cache or the CPU can put the cache into a special mode to clear certain cache cells. Either way cache flushing must be a fast operation since I/O operations are frequent. If cache flushing takes a long time then a significant performance degradation for I/O operations will occur.

Typically the following operations are necessary on the cache:

Full Flush. Completely write back all information still in the cache and invalidate all other cache cells. This insures that cache consistency and memory consistency is obtained over all of memory. However, a full flush is the most expensive and the most time consuming cache operation. Moreover, a full flush will require a refetch of currently used data into the cache when processing continues resulting in more inefficiency. It is advantageous to avoid this type of flush as much as possible.

Full Cache Clean. Simply writes back all information not yet committed to memory. Cache cleaning is useful in preparation of a DMA out operation from so far unknown locations by a device to insure memory coherence. A full cache cleaning operation is also typically very expensive but it will not result in the necessity of refetches for local variables as necessitated by the full flush.

Partial Cache Clean. A partial section of the cache may be cleaned in preparation for a DMA transfer from a specific block of memory. A partial cache clean is the most efficient cache operation since only changes in a certain range of cache cells are written to memory (writing of dirty cells).

Partial Cache Flush. A partial section of the cache is invalidated by writing out cache cells that hold data for a certain part of memory in preparation of a transfer into a this area of memory.

In addition *Cache Invalidation* commands are typically available. However, the invalidation brings with it the chance of loosing data if data structures intersect on a certain cache cell.

The issues with the 4KB page size compound the I/O problems for systems without hardware cache coherency. It is not advisable to perform cache cleaning or flushing for each 4KB page given the number of pages

⁴<http://encyclopedia.thefreedictionary.com/Bus%20sniffing>

involved. Instead one will *aggregate* as many pages as possible for I/O and then use the computational expensive full flush in preparation of multiple I/O operations. It is advantageous to aggregate I/O from different I/O subsystems and from DMA in and out operations. All necessary preparatory memory operations need to be complete before a full flush and then after the flush the hardware devices are instructed to start their work. This in turn might pose a challenge to the scheduling of I/O events on the embedded device. It could lead to other inefficiencies if the the various factors going into the decision where and when to flush are not carefully considered. Typically performance measurements are necessary to find the sweet spot for cache flushes.

If one increases the size of the cache, then the time necessary for a complete flush increases, since all cache cells have to be processed. Increasing the cache size of an embedded device therefore frequently requires the reworking of the cache flushing logic to be more efficient in order not to lose the advantage gained by a larger cache. Caches can be separated for instructions and data (Harvard Architecture)⁵ and cache operations for data exchange with the hardware can then be mostly restricted to the data cache without affecting the instruction cache thereby reducing the amount of cache flushing necessary.

These cache handling CPU penalties may be bypassed by having areas of memory that are not cached in the cache. This is particularly useful for hardware control registers. However, any access to data in those areas will be inevitably slower than in the cached areas of RAM. In cases where data is mainly handled by DMA devices this may make sense and one may even reserve a non-cached RAM area for those cases but it is balancing act and the various factors that impact performance need to be carefully evaluated.

Not having a cache coherency protocol in hardware significantly increases the complexity of device drivers and may lead to difficult trade offs in terms of speed and performance that are not easy to quantify.

7 Network I/O and Network Device Drivers

Network input and output is unique because a stream of packets is sequentially fed to a network device or read from the device. As long as the individual packets are small, cache cleaning of the memory area to be transferred may be used to avoid cache coherency issues. However, a processor that is challenged in terms of its processing speed cannot process a large number of small packets. If the embedded device has to accomplish high throughput rates then large data blocks must be passed to the network hardware and those large data blocks must be handleable with minimal support in terms of CPU processing.

Linux typically linearizes network packets and lets the driver transfer the resulting byte string to the device. Linearizing a large network packet that typically is composed of multiple segments located in the 4KB pages managed by the paging hardware (See the 4KB problem above, the improvements for Linux 2.6 Kernels also have an effect here) requires an in-memory copy operation. If the copy operation is done by the CPU then the operation can only be as fast as the speed of the CPU allows. On the other hand if one uses a hardware accelerator that bypasses the CPU, cache coherency may become an issue. Cache flushing, or the complexity of preparing descriptors for the hardware device, may eat up the benefits one has sought to obtain by the use of hardware acceleration for the copy operation.

But the Linux kernel allows a way out of this by allowing a driver to set a flag indicating that the driver is able to process the list of segments by itself. Hardware exists that will perform the linearization and the sending in one step, thereby avoiding the necessity of copying the request in memory. The hardware may at the same time also calculate the necessary checksums for the network packet. Otherwise the CPU has to perform yet another pass over the network packet to calculate the checksum which is comparable in inefficiency to the in-memory copy.

If one uses the hardware to process the list of segments then one in turn develops complexity in terms of dealing with the cache flushes. It is certainly simple to perform one complete flush for all fragments of one packet but packets are frequent. Therefore a complete flush for each packet sent may cause significant CPU load. The alternative is to have a complex cache cleaning algorithm that walks over all the segments of a packet and cleans the memory areas to be handled. However this may still be expensive in terms of CPU processing.

The aim of a network driver design for an embedded design is to limit the CPU load caused by network I/O. Any optimization that can be done in this respect will typically significantly affect the obtainable network speed. In the best scenario it will be possible to hand a group of packets to the network driver and to obtain a set of packets during reception in order to minimize the CPU load caused by frequent packet processing. This will allow one to reach maximum network throughput with a not too powerful CPU.

⁵http://en.wikipedia.org/wiki/Harvard_architecture

8 Block Driver and Block I/O

In the network device scenario sequences of varying length of bit strings have to be processed. In the block driver scenario the system passes requests for the writing or reading of byte blocks of fixed size to the driver that may be located at an arbitrary location on the device managed. These read and write requests are placed in a block device I/O queue from which the driver may pick which requests to execute next.

The challenge for the embedded device is how to process these requests in the most efficient way. Again the use of software to realize I/O of single page requests is likely to be highly inefficient. Hardware controllers for disks today typically support scatter-gather I/O that is optimized for a block device environment. The challenge of the block driver is to organize the requests in such a way that they are effectively processed by the device. In the 2.4 kernel individual 4KB blocks have to be managed by the driver which may get very inefficient for a low-powered CPU. The 2.6 kernels give already coalesced blocks of multiple times the 4KB page size to the driver and require significantly less processing by the driver. However, the 2.4 kernel block driver interface is also significantly different from the 2.6 kernels and it is not easy to port block drivers from 2.4 to 2.6 without rewriting significant portions of the driver.

The trade offs that might have to be made in terms of cache coherency are in essence the same as for the network driver. One tries to aggregate as many block requests as possible before doing a single cache flush and then start the hardware operations. Cache cleaning is not applicable to block driver design since the request sizes are so large that a complete cache flush is more effective.

9 Considerations for User Space Binaries

Embedded devices do not have much processing power and the danger with user space binaries is that CPU resources may be wasted on user space processing. The problem with many open source projects is that more and more features are added to the code as time passes. The libraries that the program depends on go through the same process and therefore programs become computationally expensive, requiring large amounts of memory and disk space. Therefore major open source tools may become difficult to use on embedded systems.

There are a variety of ways to address this issue:

Rebuild binaries with different source configuration. Binaries provided by Linux distributions are typically configured to provide maximal functionality. Not all of that functionality might be needed for the embedded device. Rebuilding the binary by configuring the source just to provide the minimal features required can provide a binary that will run much more effectively on the embedded system.

Old Source code with less features. Old releases of Linux software often require far less memory and CPU resources simply because the PCs of that area were not as powerful in terms of CPU power than today's machines. Linux became first available in 1993 and for a couple of years the kernel and user space were able to run on machines that had only 4 Megabytes of memory and were running at 16 Megahertz. Old code from those days is therefore often useful. The Debian project has a rich archive of old releases dating back to 1994. One can find version of an application one wants to use with less memory in those historical archives.⁶

Use a C library with reduced functionality. Projects like dietlibc and uclibc provide libraries utilizing far less resources than glibc. These libraries are also suitable to generate applications with limited use of resources. The source code configuration process when using these libraries typically detects limited libc functionality and does not build all features.

Busybox. The busybox project was first started for the boot disks of the Debian project. Today the busybox project provides the ability to generate a single statically linked binary that is able to run without any libraries installed. This single binary can provide the functionality of a few hundred Unix commands and allows the running of a simple system with the use of limited resources. It is frequently used for embedded systems that only have the need for minimal user space programming.

Copying of Data between Kernel Space and User Space. Network applications typically read data from the network into memory, then process that data and send it out again. These operations involve multiple in-memory copy operations that need to be avoided if possible on systems with limited CPU resources. Some applications can be configured to avoid the funneling of all data through user space through the use of the `sendfile()` system call. This is particularly important for apache and samba and may be very important for the development of customized network software.

⁶See <http://archive.debian.org>

10 Real Time OS or Embedded Linux?

There has been much talk about the use of Real Time Operating Systems for embedded systems. Various enhancements to the Linux Kernel are available that basically provide enhanced timers and the ability to schedule processes with a guaranteed response time.⁷ Most of these are incompatible with some platforms, various Linux drivers or one or the other kernel functionality and therefore are only useful in certain situations. There are also ways to configure the Linux scheduler to behave in a more predictable way (via the `sched_setscheduler()` function). In practice these rudimentary real time features are often sufficient. Other solutions require the maintenance of special kernel patches and familiarity with the special solution chosen. It is therefore often not worth in terms of the development effort required.

An alternate solution is to realize real time behavior using a coprocessor that is controlled from the primary processor. The time critical routines could be run on the bare processor with only a minimal interface to the main processor. The use of the bare processor is often advantageous in particular in embedded systems with limited resources because no operating system functionality is getting in the way (maybe just by using a few precious processing cycle) of using the full potential of the hardware. I think it is best to avoid specialized solutions and either stay with the widely used Linux kernel or go to the bare hardware. Time critical solutions require detailed knowledge of the hardware and require a very low-level interaction with the devices available. Any operating system likely an obstruction rather than help and therefore a bare CPU implementation is often easier to implement.

11 Conclusion

Embedded systems is a natural application for the Linux operating system and Linux provides a rich source environment with applications in a variety of source configurations to create an embedded system. Applications are usually easily available and scalable between different forms of hardware.

The most important performance aspects when designing an embedded system is often the cache handling. The rest of the hardware design decisions need to be driven by the decision on how to implement the control of the CPU caches in particular if the embedded device needs to be able to handle media streams today.

⁷David Kalinsky, *Basic Concepts of Real-Time Operating Systems* (LinuxDevices.com, 2004).