uClinux on COBRA5329 and general notes on MMU-less systems running Linux

Oskar Andreasson

oan@frozentux.net

Copyright © 2007,2008 Oskar Andreasson

This article is a spawn from my personal notes taken during the last couple of months working the COBRA5329 platform and the uClinux [UCLINUX] distribution. Mainly this is the work and conclusions i've drawn from trying to reduce memory usage in the uClinux kernel, and to save system resources on the platform. The main problem with shipped implementation and current uClinux implementations are the memory hungryness of the kernel and lack of a good working memory management system. From a pure hardware standpoint, this is also a discussion about the MMU-less problems and possible ways to help eliminate them.

Introduction

The goal of the original project was simple enough once I originally heard it, get a working linux 2.6 system on Coldfire 5329 evaluation board [COBRA5329KIT], and then get a graphical system running on top of it. Unfortunately the requirements was never fully apprehended by anyone in the project, and no feasibility studies had been done to begin with. Interested parties can be divided into three groups, board of directors (representing customers), software management and hardware management. Extracting and piecing together requirements has unfortunately been a part of this work. All the requirements can hence be summed up as follows:

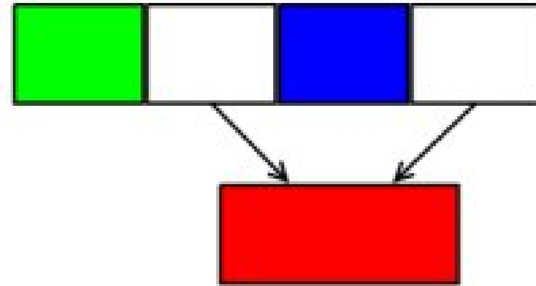
- Coldfire CPU family
- Modern GUI
- GUI easily portable between windows and host platform
- Short development time to meet TTM requirements
- Ability to run GUI and application with close to realtime responsiveness
- Extreme robustness and reliability
- Very long life expectancy of product, and availability of updates must be without doubt
- New functionality for future revisions and updates of the product must be possible

Without wallowing in the above requirements, I will dig into the software side problems that arose from these decisions on a low level. This paper discusses the hardware and software limitations that must be known to make a real decision about the viability of the Coldfire 53xx CPU's [MCF532X] in a specific application. It is very well suited for certain types of applications but in our case turned out to be far below required functionality to cover the requirements set up, mainly because of one lacking component, the MMU [MMU].

Memory

Linux memory management is the largest difference between uClinux and normal Linux. The largest problems with MMU-less processors such as the Coldfire core version 3 series, is the inability to handle virtual memory mappings. There are of course other inabilities such as memory protection, but this is minor in comparison to the virtual memory maps.

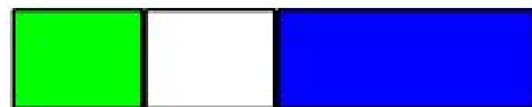
Figure 1. MMU-full memory usage



Memory usage on an MMU-full processor using SLAB. Different memory areas can be connected into virtual memory mappings using the MMU.

Modern Linux kernels has 3 different memory allocators to chose from, SLAB [SLAB], SLOB [SLOB] and SLUB [SLUB], working in accordance with slightly different principles. These principles make them more or less efficient for MMU-less systems, but in the end all three are mainly developed for MMU-full systems. In the older 2.4 kernels there is also a NPO2-allocator which was never ported forward to the 2.6 kernels, unfortunately.

Figure 2. SLAB memory splitting & usage

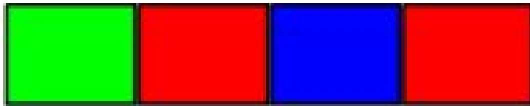


SLAB memory splitting & usage in normal circumstances. Different sized memory blocks are created at init, and prefered sizes are used/created (by mapping different areas together) for the task at hand at allocation.

MMU-less systems will work badly with the default SLAB memory allocator. SLOB is best suited while SLUB is slightly better than SLAB. Without the MMU, the cpu can not connect separate SLAB nodes, hence limiting yourself to the largest SLAB size in memory usage per process for the .data and .BSS or .text segment, or for any memory allocation for that matter. This means your application, or allocations from within your application, can never be larger than the largest allocatable block of memory, which was decided on bootup of the system by the SLAB initializer.

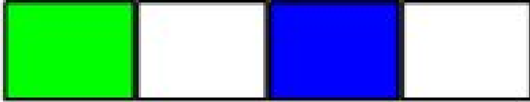
SLAB is a power of 2, allocate at init, type of memory handler. With MMU these can be connected. Without they can't, hence you are limited to the largest allocated memory chunk, at init time. At init, it assigns specific numbers of blocks with different power of 2 sizes. More of the small sizes, and the larger the chunks get, the less of them. Also worth noting, a small amount of this memory is used up for memory handling usage, hence allocating 128kb for example, might turn up with a 256kb block.

Figure 3. SLOB at start without MMU



Sample of how SLOB works without an MMU at start. Notice the red areas which have been allocated. In the next image, they are deallocated, but the blue area remains allocated.

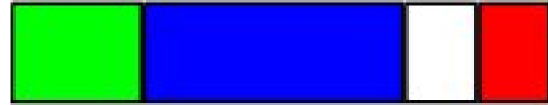
Figure 4. SLOB after deallocation without MMU



In this image the two areas have been deallocated. Unfortunately, we do not have an MMU to virtually map them together, so the next application that might want the size of both red areas can not allocate them. Defragmentation has started to wear on the system.

SLOB is non power of 2, allocate when needed, type of memory handler (K&R UNIX heap basically), with buddy freeing. Allocate memory when needed, and as much as needed, once finished, free the memory, look for boundary memory that's free and connect free area to that. This makes it possible to run larger applications on small systems. Side-effect is fragmented memory (several small objects next to eachother, one in the middle is freed and reported as such, but cant be used by larger objects. Let these allocation/deallocations go on for long enough, and you wind up with a "defragmented" memory. Former Amiga people should be very well aware of these problems.

Figure 5. SLUB example allocations



This is an example on how SLUB might allocate memory. Main point of interest, all allocations are power of 2 sized, but they are not allocated at init as in SLAB. Some of the problems still exist for SLUB as for SLAB on MMU-less systems.

SLUB seems to be a power of 2, allocate when needed, type of memory handler. Unfortunately, memory allocated seem to be unreclaimable. Ie, using memory for init applications, then trying to create a single application reusing that initial memory for a larger block, is impossible. The allocated block in this position remains the same and can not be changed. It can be reused by smaller applications/allocations, but not by larger -- unless you got an MMU of course, in which case you can virtually map it together with other memory.

The development time to simply get a working memory implementation on a lot of MMU-less systems are simply not worth the little money you save on hardware. Also, system stability takes a big hit if you have a lot of memory usage (malloc, realloc, free, etc) on sparse memory systems (<16mb ram, <16mb flash). A gathered effort is needed either by interested parties, or by the open source development community, to properly accommodate for the spartan environments of MMU-less devices, as the current implementations are simply not efficient and reliable enough.

Program sizes

For some reason, not a single binary compiled from companies get anywhere close to the sizes they state being their "stripped down size". Most companies state their library sizes in I386/x86 compilations. It might be due to the CISC architecture that these binaries get fairly small on those architectures. Another possibility is that a fairly large amount of people don't count the library sizes (it's not "part" of the application when dynamically linked).

GTK+ [GTK] stripped winded up 7.5MB (we could've put a lot more effort into stripping out unneeded code from that, and DirectFB [DIRECTFB] proved a big

problem, not supporting static linking very well). According to some sites, this should be possible in 1.5MB, but no arch mentioned.

Qtopia [QTOPIA] winded up around 4.4MB to be useful, 2.3MB with only QtCore. This is after having a lot of help from trolltech and consultants, stripping out everything not needed. According to trolltech, 1MB for maximum stripped lib, and around 2MB for a usable implementation, should be possible.

DirectFB [DIRECTFB] has serious troubles with static linking since they rely on a lot of function constructor and destructor functions, and need to dynamically load symbols etc. Hence it needed to be completely built into the end application, without serious recoding. Hence, 2.3MB to get a hello world program to work, and >4MB memory usage. DirectFB uses function constructors and destructors to dynamically load different libraries during runtime -- and since the m68knommu platform has very bad and buggy dynamic lib support -- it was not an option.

Rule of thumb, everything gets at least 2x as large as claimed.

Alternative solution for big applications

This is applicable if you want to run application from RAM for speed considerations, but dont have large enough block for it and no other memory allocators works properly.

This is a serious hack!

1. Only allocate part of memory for kernel, ie, RAMBASE + RAMSIZE smaller than reality.
2. Set FLAT_FLAG_* to some value, set value on application via modified flthdr program (in toolchain). Then modify either do_mmap() or binfmt_flat.c loader to allocate memory to hardcoded memory range, if FLAT_FLAG_* is set.
3. Specific area needs to be defined for both .text/.bss and .data segments unless FLAT_FLAG_RAM is set, in which case memory for complete blob needs to be available.
4. This is currently done by setting address directly in flat_load_file, without using do_mmap atm. Should be fixed.

This in turn creates other problems, since memory is stolen from kernel, kernel will not realize the real size of memory. Sidenote, using mem=8M etc does not work on coldfire arch, since mem size is calculated with RAMBASE + RAMSIZE. This in turn leads to smaller SLAB/SLUB allocations, and since you're application is big, you might want large mallocs as well.. which will not work.

Other stability problems with these changes has been noted, not sure why at this moment or if it was at all related since I later discovered that the power supply used during these tests was malfunctioning and giving small power fluctuations, enough to reset/deadlock the CPU. Also a problem with above solution, gdb will not work. Gdb needs to do things inside the code (set TRAP's, etc), and uses mm functions, which are not aware of the memory (we hardcoded it "not to be memory").

SLOB mremap brokenness

SLOB reverts to using a lot of SLAB functions, one of these,

```
kobjsize(const void *objp)
```

, can not be used on objects not of SLAB origin. This is broken in every kernel between 2.6.17 and 2.6.22 that i've tried. A flawed fix is to comment out/remove all the current code in kobjsize and replace it with

```
return
(ksize(objp));
```

. There are hints at a proper fix out there, but I didn't spend time finding it.

See http://readlist.com/lists/uclinux.org/uclinux-dev/1/5622.html. I have yet to find more in this, my searches are wild and probably in the wrong direction but ended quick since it was fairly easy to hack once the problem was understood.

Fuzzy Logic SLOB

This is a minor idea I had on a new SLAB/SLOB replacing memory allocator. The actual implementation of this was out of the scope of the original project, and finding both the time and experience to actually implement it is probably out of any possible timeframe for me at the moment. It would require heavy changes in gcc, glibc and the kernel memory management to work. This memory allocator is particularly interesting on MMU-less systems as it could possibly solve some of the problems of memory defragmentation, but it also relies on some heavy calculations which might be less optimal for small systems, and it also relies on the non-virtualness of the MMU-less systems, but which could possibly be alleviated on MMU-full systems as well. For further discussion regarding this topic, please contact me and I will try to explain what I mean.

The idea came while doing this work, and simultaneously reading a book on Fuzzy Logic. Let the kernel move the applications and their allocations between different memory regions. To do this, a few fundamental changes are required. Kernel must have knowledge of all taken memory allocations. Kernel must have knowledge of all pointers to memory allocations. Applications must have knowledge that kernel needs knowledge about pointers to memory allocations. All memory pointers must be reported to kernel by application.

With this information, the kernel can maintain a list of references to all memory references (a void **ptr), as well as a list of all current memory allocations. To move an allocation, go into uninterruptable mode, move the chunk of memory to the new location, go through list of pointers find the ones inside old memory chunk, change them to the new memory area, leave uninterruptable mode. This achieves a memory handler that can be defragmented, but requires a memory handler that actually implements it. Enter "Fuzzy Logic SLOB", in lack of a better description.

The basic idea is to have moving subsets of memory, each being more or less suitable for different types of memory allocations. A small memory allocation would be better suited for a subset with a high fragmentation and preferably a high usage. A larger memory allocation would be better suited inside a low fragmentation and low usage memory subset. Having the size of each subset move facilitates changing natures in the system, letting the system adapt to the kind of workload it copes with. There may be as many subsets as needed, covering different memory address zones and handling slightly varying requirements.

As an allocation is made, the most suitable subset is evaluated, and memory allocator for the given subset is tested, if no memory was found, try the next subset, and so forth. If no sufficiently large memory area is found, shuffle memory around according to some algorithm and try to free up a sufficient address area for the allocation. Of course, when this happens, all pointers inside applications must be updated.

There are several severe drawbacks of this memory handler, but in very specific cases it triumphs over other solutions. For example, the cases where having an MMU would allow for better memory management, but not enough to warrant the higher price. In large mass production, even a single dollar per unit adds up to millions per year. As it currently stands, you would either have to get an MMU or a very large RAM to handle these circumstances in Linux. Having the option of very slow operation, but decent memory handling at a very cheap price will be worth it in several very large mass production scenarios.

Cross-compilers

Some problems that where very hairy to debug, and some very simple, stemed from the gcc/uclibc toolchains that we used. None of the newer toolchains seems to have been properly inserted/tested by the uClinux community, except for CodeSourcery. Their pre-built products are extremely nice from what I have heard, but the build chains are next to impossible to work with. Hence, we started out with an old m68k-uclinux-gcc-3.4.0 toolchain that was previously used on a mcf5272 platform, but some of the problems encountered made us question this decision and we went for a m68k-uclinux-gcc-4.1.1 in the end. It worked fairly well with some minor details.

First off, and most major problem, it does not handle inlined functions properly without -O2 flag. Code is lost, without a trace, nor error/warning message. Main problem in linux kernel seems to be the `next_thread` function in `include/linux/sched.h`. Without de-inlining this function, kernel will not boot properly. A lot of time had to be spent with the lauterbach debugger before I realized what was happening. I have not done a terrible lot of research on the internet on this bug, so I don’t know if this is a known bug or

not. Other changes includes:

- PATH was incorrectly set in build script, and hence binutils where not available at early stage in build process.
- uClibc configurations where not copied properly from/to the correct place in filesystems during build, and changes to configuration was required to make it work on the MCF5329 platform. WCHAR support was a bit of a mess to get since it required setting "Manuels Hidden Warning messages" to even show up in the uClibc configuration.
- Gcc would not build by default on our build system, nothing major. Mainly defines and paths needing fixes. One of the bigger problems, except for the above mentioned -O2 and inlining problem, was some missing support for the ror.w assembler code. This has been fixed in newer releases of gcc and was easily dealt with.
- Added libxml2 to buildprocess since it is needed in the end product. This also added a stage8 in the build scripts.
- Added support for making "load to special ram" binaries with the flthdr and elf2flt packages, as defined in previous part of this article. These application changes where incredibly simple to work out.

To summarize the toolchain part of this project, once a fairly recent and modern gcc was chosen for the project, no insurmountable or unexpected problems really arose except for the inlining bug which was possible to get around once it was known. An even more modern gcc would have been preferable, but because of the lack of properly setup chains easily accessible, we stayed with the 4.1.1 toolchain.

Linux XIP kernel

XIP stands for eXecute In Place [XIP]. In normal execution, the application (or kernel) is moved to RAM memory and extracted to create enough size for each application segment (.bss, .data, etc). XIP makes it possible to execute the application from their actual location on harddrive/flash/storage device, only copying/creating the segments that actually needs to be moved. This saves a bit of RAM. The coldfire platform we've worked with has 16MB flash and 16MB sdram, possibly 32MB further down the road. To save some of this, a XIP kernel was implemented.

XIP is fairly common on coldfire platforms and worked fairly well with old 2.4 kernels on mcf5272 platforms, not so on the mcf5329 with a 2.6 kernel. The kernel contains a .text, .data, .bss and .init segments and on .init's end, ROMfs is located. The .bss is empty until the binary is loaded (and actually zero sized until it is extracted).

Normal runtime moves the whole binary (including filesystem!) to ram and then executes, this is actually done by dBUG, which then enters the kernel at the _start symbol, which is located at a specific memory address (0x40020000 on MCF5329 Cobra card). dBUG needs to be rewritten (in the end we will probably write a new bootloader) to take care of this.

XIP execution executes from flash directly, not RAM. Only the parts that must be in ram is moved up to ram, the rest resides in flash. the bootloader starts executing the kernel directly at its flash location, and after this, the _start function must do specific operations before it can actually start the kernel. .text should stay in flash (non-alterable code) while .data and .init must be moved up to RAM (it can be altered) and finally a zero'd out .bss must be created in RAM. You will also need to retain some kind of symbol that points at kernel end in flash, where the ROMfs is located, otherwise your kernel will once again try to load it from RAM.

Another problem seems to be that MTD [MTD] probing can not be done on the same device that the code is currently executing from. Look in the MTD init code. I managed to get around this so far by telling it that the ROM MTD is actually RAM MTD, which is a rather ugly hack, and which has proven to cause several mysterious hangs and crashes. A better, but still not very good hack, would be to hardcode the abilities in the probing functions instead. The final and best way would be to enable the probing code to either probe the device we're executing from (from rumours this is not possible due to hardware limitations), or move the probing code to real RAM, and then execute it (ie, move the code to .data segment perhaps in the ld-scripts).

Conclusion

Conclusions are hard to draw from the problems that we've run into so far in this project. In my honest opinion, I don't think the hardware platform with the uCLinux choice is stable enough to warrant a recommendation for anyone else than hobbyists. I've

long been a pro-Linux user, admin and developer, but this forray has seriously put some very bad lighting on Linux as a system to me.
The community support that Linux prides itself in has been less than optimal, close to impossible to get, the same goes for commercial support. The state of the memory management is absolutely horrible for the hardware/application requirements, and there is next to no support from the general linux developers to the people who actually wants to make Linux a viable option on MMU-less systems from what I've seen.
That said, I have full understanding of why or how this has happened. The development kits from freescale and its ilk is horribly expensive (in comparison to other embedded platforms), hence out of reach for a lot of part time hobbyists. The embedded community is small and most all developers seem to make their own little hacks that don't get caught and put into the default systems. Also, Linux in general seems to grow rapidly in desktop and large computational services, and those are fairly far from embedded solutions, I'd go as far as saying they are diametrically opposites of each other. Making a OS that works for both these opposites would be a gigantic feat in my humble opinion.
What should be done? I'm not quite sure. I think Freescale must understand that they need a cheap and easily available development software platform, that is open source and well documented and taken care of, for the coldfire to grab the general Linux populace attention. A really good starting point would be to look at the gumstix platform, which is brilliant in my opinion. The coldfire processors are for cheap projects in comparison to the Xscale used in gumstix, hence the development kit(s) should be lower cost than the gumstix to fully get the user and developer base required. That said, I'm not totally sure there is a large enough market for this kind of product, but I hope so.
This project was a system critical hardware, that must not fail, and a lot of other must. The hardware platform together with the chosen software platform proved insufficient to handle the requirements, which is very unfortunate. The project has now moved on to a larger and more expensive hardware platform (Atmel ARM based), which will cost the company quite a bit in production -- The product is estimated to sell in quantities of perhaps 5-20 000 units per year for a duration of 10-15 years forward, adding an extra production cost of 10 USD for a larger processor due to bad software support makes quite a hefty sum in the end.

On our part, the job of analyzing the demands and requirements of the platform should have been done earlier, as well as a feasibility study on the combination of the hardware and software. The opposite side nature of the requirements set on the project has also caused some severe conundrum for the development team, and the priorities should've been cleared out at an earlier stage as well. This would have allowed us to either say straight off that it will not work, or walked away with a less futureproof version that at least works for now, but perhaps not on the next platform.

Bibliography

Papers

[OPDEN2006] Michael Opdenacker, *Embedded Linux Optimizations*, http://www.embedded-kernel-track.org/2006/michael_opdenacker_optimizations.pdf.

[OPDEN2007] Michael Opdenacker, *Embedded Linux kernel and driver development*, http://fmgroup.polito.it/ cabodi/dida/sp/embedded_linux_kernel_and_drivers.pdf.

[AN3408] Yaroslav Vinogradov and Roznov pod Radhostem, *Building a Sample CGI Application - for the uClinux-Targeting ColdFire MCF5329 Evaluation Board*, http://www.freescale.com/files/32bit/doc/app_note/ AN3408.pdf?fsrch=1.

[EMBEDDEDMEDIA] http://free-electrons.com/doc/embedded_linux_ multimedia.odp.

Hardware

[GUMSTIX] *Gumstix Official homepage*, http://www.gumstix.com/.

[GUMSTIXORG] *Gumstix Community*, http://www.gumstix.org/.

[MCF532X] *MCF532X: V3 ColdFire Microprocessor with LCD driver, Ethernet, USB and CAN*, http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MCF532X&nodeId=0162468rH3YTLC00M92076.

[COBRA5329KIT] *DragonFire - COBRA5329KIT*, http://www.ebv.com/en/ products/highlights/ freescale_dragonfire.html.

Web resources

[UCLINUX] *uClinux - Embedded Linux/Microcontroller Project*, http://www.uclinux.org/.

[UCDOT] *uCDot newssite*, http://www.ucdot.org/.

[SLOB] Matt Mackall, *slob: introduce the SLOB allocator*, http://lwn.net/Articles/157944/.

[SLAB] *Slab allocation (wikipedia)*, http://en.wikipedia.org/wiki/Slab_allocation.

[SLUB] Christoph Lameter, *Short users guide for SLUB*, http://www.mjmwired.net/kernel/Documentation/vm/slub.txt.

[DIRECTFB] *DirectFB Official Homepage*, http://www.directfb.org/.

[MMU] *Memory Management Unit (wikipedia)*, http://en.wikipedia.org/wiki/Memory_management_unit.

[MTD] *Memory Technology Device (wikipedia)*, http://en.wikipedia.org/wiki/Memory_Technology_Device.

[XIP] *eXecute In Place (wikipedia)*, http://en.wikipedia.org/wiki/Execute_in_place.

[GTKDFB] *GTK on DirectFB*, http://www.directfb.org/wiki/ index.php/Projects:GTK_on_DirectFB.

[GTK] *The GTK+ Project*, http://www.gtk.org/.

[QTOPIA] *Trolltech Qtopia*, http://trolltech.com/products/qtopia.

[QTOPIA2] *Qtopia Community*, http://qtopia.net/.

[GFXQUICKREF] *Embedded Linux Graphics Quick Reference Guide*, http://www.linuxdevices.com/ articles/AT9202043619.html.