Introducing the Advanced XIP File System

Jared Hulbert
Numonyx
jaredeh@gmail.com

Abstract

A common rootfs option for Linux mobile phones is the XIP-modified CramFS which, because of its ability to eXecute-In-Place, can save lots of RAM, but requires extra Flash memory. Another option, SquashFS, saves Flash by compressing files but requires more RAM, or it delivers lower performance. By combining the best attributes of both with some original ideas, we've created a compelling new option in the Advanced XIP File System (AXFS).

This paper will discuss the architecture of AXFS. It will also review benchmark results that show how AXFS can make Linux-based mobile devices cheaper, faster, and less power-hungry. Finally, it will explore how the smallest and largest of Linux systems benefit from the changes made in the kernel for AXFS.

1 Filesystems for Embedded Systems

1.1 Why not use what everyone else uses?

Embedded systems and standard personal computers differ a great deal in how they are used, designed, and supported. Nevertheless, Linux developers tend to think of the Flash memory in an embedded system as the equivalent of a hard disk. This leads many embedded Linux newbies to ask, "Can I mount ext2 on the Flash?" The simple answer is yes. One can mount ext2 on a mtd-block partition in much the same way that one can bathe in a car wash. However, for both the car wash and the MTD, this is not what the system was designed for and there are painful consequences—however, they get the job done… well, sort of.

There are a few key differences between filesystems used in a personal computer and those used in embedded systems. One of these differences is compression. Many filesystems used in embedded systems support compression of file data in 4KB–128KB blocks. Cost, power, and size limitations in embedded systems result in a scarcity of resources. Compression helps to relieve some of that scarcity by allowing the contents of a usable rootfs image to fit into a reasonably sized Flash chip. Some embedded filesystems make use of the MTD device API rather than a block device API. Using an MTD device allows the filesystem to take advantage of special characteristics of Flash and possibly avoid the overhead of the block device drivers. A third difference is that a read-only filesystem is perfectly acceptable for many embedded systems. In fact, a read-only filesystem is sometimes preferred over a writable filesystem in embedded systems. A read-only filesystem can't be corrupted by an unexpected loss of power. Being read-only can offer a bit of security and stability to an embedded system, while allowing for a higher performance and space-efficient filesystem design.

Linux filesystems that are well suited to the needs of embedded systems include CRAMFS, JFFS2, SQUASHFS, YAFFS2, LOGFS, and UBIFS. Only two of these filesystems have been included in the kernel. CRAMFS is a very stable read-only filesystem that supports compression and mounts from a block device. There is an out-of-tree patch set for CRAMFS which enables it to run XIP with no block device. If you need to write data on Flash, the only option in the kernel is the mature JFFS2. Like JFFS2, YAFFS2 requires an MTD device, but does not provide compression like JFFS2. YAFFS2 has been around for several years, but getting it into the Linux kernel does not seem to be a priority for the developers. SQUASHFS is over 5 years old and is included in nearly every distribution. While the developer has made attempts to get it pushed to mainline, those attempts have been sidelined by a surprising amount of resistance. As an improvement over CRAMFS, SQUASHFS is capable of creating larger filesystems at higher compression ratios. LOGFS and UBIFS are projects with the same goal, providing more scalable, writable Flash filesystems for growing NAND storage in embedded systems. Both support compression and both are trying hard to be included in the kernel by 2.6.26.

1.2 The inconvenient filesystem

The linear XIP CRAMFS patches have proved useful in many small Linux systems for over 8 years. Unfortunately, the patched CRAMFS contains calls to low-level VM functions, a mount option to pass in a physical address, and modifications to the virtual memory code. The main cause of this hubris is that this patched CRAMFS doesn't fit the filesystem paradigm and therefore doesn't fit the infrastructure. The result is an ugly hack and a maintenance nightmare. The patch set broke badly almost every year due to some change in the kernel, because it messed with code which filesystems have no business touching. Not only is XIP CRAMFS hard to maintain and port, it also has serious limitations. CRAMFS only supports ∼256MB image sizes and the maximum file size is 16MB. Notwithstanding these limitations, the linear XIP patches to CRAMFS have been included in most embedded Linux distributions for years. Variations on linear XIP CRAMFS ship in millions of Linux-based mobile phones every year.

Embedded Filesystem Summary

filesystem	compress	MTD/block	writable	XIP	in-kernel
CRAMFS	✔	block	×	×	✔
JFFS2	✔	MTD	✔	×	✔
YAFFS2	×	MTD	✔	×	×
SQUASHFS	✔	block	×	×	×
LOGFS	✔	MTD	✔	×	soon
UBIFS	✔	MTD	✔	×	soon
XIP CRAMFS	✔	×	×	×	×

2 Changing the Filesystem Paradigm

2.1 The current filesystem paradigm

Performing any real operations on data requires the data to be in memory. Executing code also requires that it be in memory. The role of the filesystem in Linux is to order data into files so that it can be found and copied into to memory as requested. In a typical personal computer, a Linux filesystem copies data from a hard disk to RAM. The Linux kernel filesystem infrastructure assumes that data must be copied from a high-latency block device to the low-latency system RAM. While there are a few minor exceptions, this is the rule. This is the basic filesystem paradigm driving the architecture of the Linux virtual filesystem.

A typical embedded system today might have an ARM processor with some embedded Flash memory and some RAM. The filesystem would copy data from Flash memory to RAM in this case. While Flash memory has much faster read latency than a hard disk, the basic paradigm is the same. Linux developers tend to think of the Flash memory in an embedded system as the equivalent of a block device like a hard disk. This fits the filesystem paradigm built into the kernel, therefore few kernel developers care to investigate whether the paradigm fits the realities of the hardware.

2.2 Why XIP?

What is so useful about the XIP-patched CRAMFS that has prompted it to be haphazardly maintained out-of-tree for nearly a decade? What is it about these patches that make them so hard to reconcile with the kernel? The answer to both is eXecute-In-Place, or XIP. Executing a program from the same memory it is stored in is usually referred to as XIP. As code must be in memory to be executed, XIP requires a memory-mappable device such as a RAM, ROM, or a NOR Flash. NAND Flashes are not memory-mapped and thus not suitable for XIP; they are more like block devices than RAM.

XIP is common in RTOS-based embedded systems where the memory model is very simple. In a RTOS, to add an application from Flash into the memory map, the designer need only specify where in the Flash the application is, and link the application there during the build. When the system is run, that application from Flash is simply there in memory, ready to be used. The application never needs to be copied to RAM.

A Linux application in Flash is a file that would be contained in a filesystem stored on the Flash. To get this application from Flash into a memory map, individual pages of the application would be copied into RAM from Flash. The RTOS system would only require the Flash space necessary to contain the application. The Linux system would require the Flash space necessary to store the application, and RAM space to contain the application as it gets copied to RAM. The Linux filesystem paradigm treats the Flash as a block device. The Flash-as-block-device paradigm overlooks the memory aspect of the Flash *memory*. The result is wasted resources and higher cost.

If we have in our embedded system a Flash that can be used as memory, why not use it as such? When such

Flash is used as memory, the system can use less memory by removing redundancy as explained above. Using less memory results in reduced cost, which is always a priority for consumer electronics. Reduced memory also reduces power, which increases battery life. Performance can also be improved with XIP. If an application does not need to be copied to RAM nor decompressed—only pointed to—to be used, the paging latency is drastically reduced. Applications launch faster with XIP. Where it can be used, XIP is a great way of improving on system cost and performance. For years the only option was to depend on the limited and hacked linear XIP CRAMFS patches.

2.3 Expanding the paradigm

The first step toward consolidating the XIP features of CRAMFS used in the smallest of Linux system into the kernel came from an unlikely source, one of the largest of Linux systems. As part of the 2.6.13 merge, the s390 architecture tree introduced a block driver for `dcss` memory that extended the block device to include a `.direct_access()` call. This new interface returns an address to a memory region that can be directly accessed. It allows for XIP from memory which is posing as a special block device. To complete the system modification to ext2 were made, and the functions in `/mm/filemap_xip.c` were introduced to allow data on this `dcss` memory to be used directly from where it was stored. The s390 architecture users find this feature very useful because of the way their systems allow for Linux virtualization. Because many virtual systems were sharing a root filesystem, requiring each system to maintain copies of important files and code in a page cache when it was already accessible in a shared memory device would be a huge waste of resources.

With these changes to the kernel, the filesystem paradigm changed a bit. Data no longer had to be copied from a block device into RAM before being used; the data that is stored in a special memory device can be mapped directly. While the embedded Linux world continued to fumble with the hacked CRAMFS patches, the mainframe Linux developers laid the foundation for merging XIP into the kernel.

3 Why a new filesystem?

3.1 The Problems

In order to take advantage of the memory savings and performance benefits that XIP has to offer, Linux needed a few more tweaks and a filesystem. Although the `/mm/filemap_xip.c` infrastructure was a step in the right direction, it did not address all the problems with adding XIP functionality for embedded systems. The changes introduced by `/mm/filemap_xip.c` added a new function, `get_xip_page()`, to `struct address_space_operation` that a filesystem was supposed to use to pass a `struct page` for the memory that was to be inserted into a process's memory map directly. In an embedded system, the memory that is to be passed is Flash, not RAM, and has no `page` associated with it. The way the XIP CRAMFS patches handled this was to call `remap_pfn_range()`. This was one of the causes of the maintenance problems with the patches. Because the API for doing this was intended for limited use in drivers and internal memory management code, not for filesystem interfacing, it changed relatively often. A solution would need to be found that modified the infrastructure from `/mm/filemap_xip.c` with the functionality enabled by calling `remap_pfn_range()` directly.

With no `struct page` to leverage in mapping memory, the kernel would need the physical address of the memory to be mapped. The XIP CRAMFS patches solved this by requiring a *-o physaddr=0x...* at mount. While this approach works, it violates some of the layering principles Linux developers try to enforce. This approach required the filesystem to deal with hardware details, the physical address of a memory device, which are supposed to be handled by driver levels. There were also conflicts with the `ioremap()` call in the filesystem which mapped these physical addresses into kernel addressable virtual addresses. There were some rather important architecture-specific `ioremap()` optimizations controlled by `#ifdef`, creating more confusion and calling more attention to the layer violation.

Analyzing and comparing systems with and without the XIP CRAMFS patches lead to a discovery. Under some circumstances, XIP CRAMFS would indeed save RAM, but it would do so spending more space in extra Flash than was saved. One secondary reason for this mismatch was that XIP CRAMFS had a rather inefficient way of mixing XIP and and non-XIP files. XIP files must be aligned on Flash at page boundaries in order for the memory to be directly inserted into the memory map. XIP CRAMFS left possible alignment holes at the beginning and end of each XIP file. The major cause of this skewed exchange rate was that XIP CRAMFS uncompressed entire files even if only a small part of that file was ever mapped. In a non-XIP system, only the pages that actually did get mapped would be copied into the RAM. To realize true memory savings, a solution would need to be able to identify and XIP at a page granularity rather than a file granularity. Unfortunately the kernel internals capable of inserting physical pages, not backed by a `struct page`, did not allow page-by-page granularity.

If any effort was to be expended on creating a mainlineable XIP filesystem solution, one could not ignore the limitations of CRAMFS. 256MB is painfully close to today's largest NOR Flash chip and is many sizes smaller than today's NAND Flash chips. Even SQUASHFS (which supports 4GB filesystems) was criticized as being "limited" by kernel developers. Simply rearchitecting the CRAMFS patches would not produce a sufficiently scalable solution. Even using SQUASHFS as a starting point might be viewed as not scalable. SQUASHFS would also need to be modified to use MTD devices. JFFS2 could also be viewed as not scalable. It should also be noted that JFFS2, having a writable architecture, would introduce many additional complexities if used as the basis for an XIP filesystem. We decided the best solution was to create a filesystem designed from the ground up to support XIP.

Obstacles to extending existing filesystem for XIP

1. No `struct page` for `/mm/filemap_xip.c`
2. Physical address not provided by drivers
3. XIP/compression on page granularity not supported
4. Existing filesystems "limited" or poor fit to application

3.2 Removing Barriers

As we looked at our options to enable XIP with a sustainable solution, it became obvious that we needed to address the remaining issues in the kernel infrastructure. In order to remove the `struct page` dependency in `/mm/filemap_xip.c` we worked with virtual memory developers as well as the developers of the s390 architecture. Amazingly, the s390 developers were as excited as we were to remove the `struct page` dependencies from the `/mm/filemap_xip.c` for much the same reasons we had. In both the s390 architecture and the classic ARM-plus-Flash embedded system, the XIP memory is memory, but really doesn't want to be thought of as system RAM. Adding a `struct page` increases RAM overhead, but did not deliver any true benefit to our systems. Only in the Linux community do you find "big iron" and embedded systems developers working toward the same goal. The end result is a set of patches that is in the `-mm` tree as of this writing, hoping for a 2.6.26 merge.

Mapping non-`struct page` memory into memory maps requires that the filesystem be able to get the physical address that the virtual memory layers require. The target system, an ARM processor with Flash memory, would be able to have an MTD partition for the filesystem image to reside in. Using the little-used `mtd->point()` would give the filesystem a kernel-addressable virtual address to the image on Flash. While it is tempting to try a `virt_to_phys()` conversion, this simple approach doesn't work for our target architecture. The only place that reliable information about the physical address of Flash resides is in the MTD subsystem. Mounting to an MTD and then getting the physical address from the MTD seems reasonable. However, the MTD interface didn't provide an interface to get the physical address. The MTD developers decided the best way to get the physical address was to extend the `mtd->point()` to include a virtual and a physical address. There is a patch that has been signed off by many key MTD developers and will hopefully be merged in the 2.6.26 window.

Allowing control over the decision to XIP or compress pages at a page granularity requires both a new filesystem architecture and a change to the kernel. The patches required to do this are included with the pageless XIP recently added to the `-mm` tree. At issue were the mechanisms available to insert physical addresses in the form of a page frame number, or *pfn*, into process memory maps. Inserting a pfn with no `struct page` required the use of the `VM_PFNMAP` flag. The `VM_PFNMAP` flag makes assumptions about how the pfns are ordered within a map. These assumptions are incompatible with enabling a page granularity for XIP. The `VM_MIXEDMAP` patch allows a process's memory to contain pfn-mapped pages and ordinary `struct page` pages in an arbitrary order. Allowing pfn-mapped and `struct page`-backed pages to coexist in any order allows a filesystem to have control over what parts of what file are XIP, and which are copied to the page cache.

4 Architecture

4.1 Design Goals

Once we decided that none of the existing Linux filesystems was likely to be easily extended to have the feature set we required, development of the architecture for the Advanced XIP File System began. The target application we had in mind was in mobile phones. Many phones use CRAMFS or SQUASHFS; therefore, many of the features will overlap with these existing filesystems. Being read-only and having limited time stats is acceptable. We need to improve on the size limitations built into CRAMFS and SQUASHFS by creating a 64-bit filesystem. Compressing in greater than 4KB chunks like SQUASHFS should also be enabled. The new filesystem should be able to mount from block devices like the legacy filesystems, but it should also mount directly from a MTD device. One new thing that we needed to add was the ability to mount with part of the image on a memory-mapped NOR Flash chip, while the rest of the image in on a NAND-style Flash chip.

4.2 Feature List

1. Basic Attributes
   - 64-bit
   - read-only
   - designed with embedded needs in mind
2. Compression
   - 4KB–4GB compression block size
   - page-by-page uncompression map for XIP
3. Flexible Mount
   - MTD (NAND/NOR)
   - block device
   - split across XIP NOR and non-XIP NAND
4. Tools
   - GPL mkfs.axfs
   - Supported image builder available

4.3 Profiling

Having the capability to decide whether to use XIP on individual pages allows the system designer to make a more cost-effective system, in theory. The obstacle in making this work in practice is deciding the right pages to XIP. The way we decided that made sense to us was to measure which pages in a filesystem are actually paged in. We chose to have a profiler built into the AXFS filesystem driver. The profiler records each time a page from a file in an AXFS filesystem is faulted into a non-writable map. After important use cases, the result can be read out of a */proc* file and then the profile buffer can be reset by writing to that same */proc* entry. There is a Kconfig option to allow the profiler to be compiling in or out of the driver. To profile a system, the system designer takes the following steps:

1. profiler is compiled in
2. system is booted with an AXFS image
3. important use cases are run
4. profile is extracted from */proc*
5. profile is fed back into the image builder
6. profiler is compiled out
7. optimized AXFS image is loaded into system

4.4 Mount Options

Figure 1 shows how the same image can be mounted either on a single device or split across two devices. This is to allow a system designer maximum flexibility in optimizing systems for cost. A typical use for this device-spanning capability is to allow an image to span a NOR-type Flash and a NAND Flash. Device spanning is only permitted if the first device is directly memory-mappable. Any XIP regions of the AXFS image would

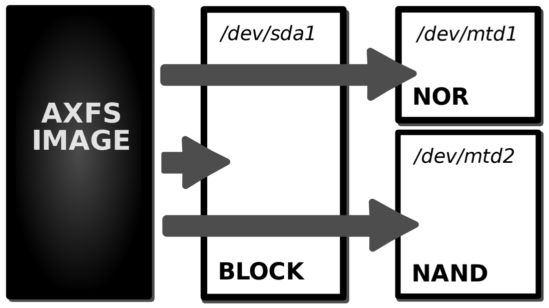

Figure 1: Mounting Details

need to reside on a NOR Flash. However, there is no reason why the compressed page regions would need to be in NOR. As NOR Flash is usually more expensive per bit compared to NAND Flash, placing compressed page regions in NAND Flash makes economic sense. It is not uncommon to have a large amount of NAND Flash in a mobile phone today. The amount on NAND Flash is often driven by media file storage demands rather the size of the root filesystem.

XIP CRAMFS would require a large enough NOR Flash be added to the system for the entire root filesystem image. Such a system could not take advantage of two facts: first, only a part of the image requires the XIP capabilities of the NOR Flash; and second, there is a large NAND Flash available. With AXFS, the NOR Flash can be as small as the XIP regions of the AXFS root filesystem image, with the rest of the image spilling over into the NAND. The reason this can lead to cost savings is that each memory component—the RAM, NOR Flash, and NAND Flash—can be sized to minimum-ration chip sizes. If RAM usage is just over a rational chip size, but there is room in the NOR Flash, the designer can choose to XIP more pages. If the NOR Flash usage is over a chip size, but there is free RAM, pages can be compressed and moved to NAND Flash to be executed from RAM.

This flexibility also lowers risk for the designer. If applications need to be added or turn out to be larger than planned late in the design cycle, adjustments can be made to the contents of RAM and NOR Flash to squeeze the extra code and data into the system while retaining performance. With a traditional system, unexpected code size means unexpectedly high page cache requirements. This leads to more paging events. As system performance is sensitive to paging latencies, more code will certainly lead to lower system performance. When this happens with an AXFS system, any free space in NOR Flash can be exploited to to absorb the extra code or to free up RAM for data.

4.5 Format

The AXFS on-media format is big-endian and has three basic components: the superblock, RegionDescriptors, and Regions. Essentially the superblock points to the RegionDescriptors, which in turn point to Regions (as shown in Figure 2). There is, of course, a single superblock, many RegionDescriptors, and many Regions.

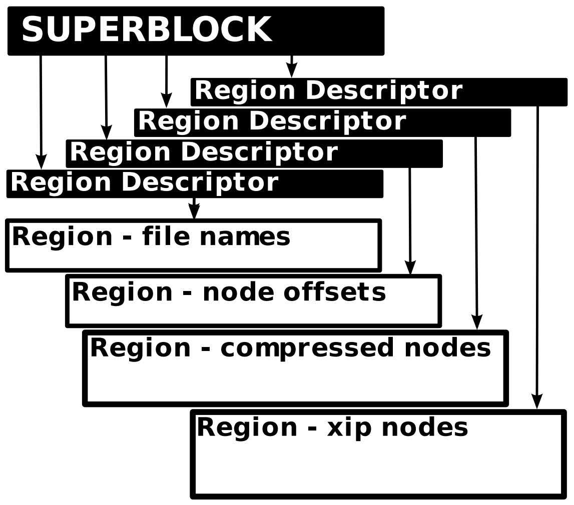

Figure 2: On-media Format

The superblock contains filesystem volume-specific information and many offsets, each pointing to a separate RegionDescriptor. Each RegionDescriptor then contains an offset to its Region. A Region can contain many different kinds of data. Some Regions contain the actual file data and one contains the filename strings, but most Regions contain ByteTables of metadata that allow the files to be reconstructed.

ByteTables are how most numbers are stored in AXFS. It is simply an array of bytes. Several bytes must be "stitched" together to form a larger value. This is how AXFS can have such low overhead and still be a 64-bit filesystem. The secret is that in the AXFS driver, most numbers are unsigned 64-bit values, but in the ByteTables, each value is only the minimum number of bytes required to hold the maximum value of the table. The number of bytes used to store the values is called the depth. For example, a ByteTable of offsets into the Region containing all the file name strings could have many depths. If the strings Region is only 240 bytes long, the depth is 1. We don't need to store offsets in 8-byte-wide numbers when every offset is going to be less than the 255 covered by a single byte value. On a strings Region that is 15KB in size, the depth would be 2, and so on. The ByteTable format makes it easy to support large volumes without punishing the small embedded systems the design originally targeted.

A Region in AXFS is a segment of the filesystem image that contains the real data. The RegionDescriptors on media representations are big-endian structures that describe where a given Region is located in the image, how big it is, whether it is compressed, and for Regions containing ByteTables information, information about the depth and length of that ByteTable.

Several ByteTable regions are dedicated to inode-specific data such as permissions, offset to the filename, and the array of data nodes for files or child inodes for directories. Those are fairly straightforward. The data nodes prove a little more complex. A file inode points to an array of page-sized (or smaller) data nodes. Each node has a type (XIP, compressed, or byte-aligned) and an index. The XIP case is the simplest. If the node type is XIP, the node index becomes the page number in the XIP region. Multiplying the index by `PAGE_SIZE` yields the offset to the XIP node data within the XIP Region.

A node of type *byte-aligned* contains data that doesn't compress and is a little more complicated to find. This exists because data that doesn't compress is actually larger when run through a compression algorithm. We couldn't tolerate that kind of waste. The node index becomes the index into a ByteTable of offsets. The offset points to the location within the byte-aligned Region where the actual data is found.

The compressed node type is the most complicated to find. The node index for a compressed node is used as the index to two separate ByteTable values, the *cblock* offset and the *cnode* offset. A cblock is a block of data that is compressed. The uncompressed size of all cblocks is the same for a given filesystem, and is set by the image builder. A cnode is a data node that will be compressed. One or more cnodes are combined to fill a cblock and then compressed as a unit. The compressed cblocks are then placed in the compressed Region. The cblock offset points to the location of the cblock containing the node we are looking for. The cblock is then uncompressed to RAM. In this uncompressed state, the cnode offset points to where the node's data resides in the cblock.

5 Benchmarks

5.1 Benchmark Setup

The benchmarks were run on our modified "Mainstone 2" boards. The kernel was only slightly modified to run on our platform and to include AXFS. The root filesystem was a build of Opie we created from OpenEmbedded about a year ago. It is a full PDA-style GUI, and we added a few applications like Mplayer and Quake.

- PXA270 Processor
  - 520 MHz (CPU)
  - 104 MHz (SDRAM bus)
  - 52 MHz (NOR flash bus)
- Linux-2.6.22
  - xipImage
  - `CONFIG_PREEMPT=y`
  - MTD updated to Sept 25 git pull
  - `mem=24MB` in kernel commandline
- Opie root filesystem
  - OpenEmbedded
  - about one year old

5.2 Performance

Rather than demonstrate meaningless raw throughput numbers, we wanted to use a real-life scenario that shows a difference. The easiest way to show how AXFS can improve performance is by showing how fast it can be at launching applications. We reduced the available RAM to simulate the memory pressure present in a cost-sensitive consumer electronic device. Figure 3 shows the combined time it took to start up several applications in seconds for each filesystem. Once these applications (video player, PDF viewer, web browser) launched, they

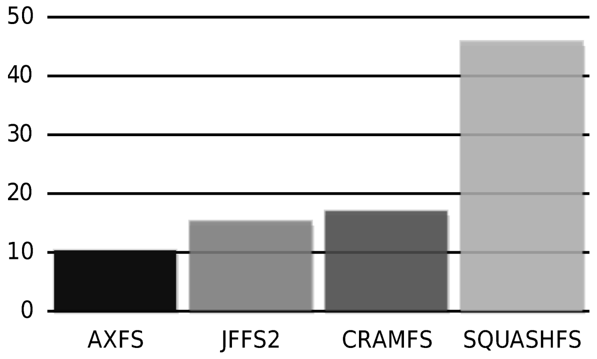

Figure 3: Aggregate application launch (s)

performed the same on each platform as far as we could measure.

Figure 4 shows the effect of memory pressure on the various filesystems. This shows the time it took to boot to the Opie splash screen with the amount of RAM varied via `mem=`. As the memory pressure builds, you can see that the performance is very steady on AXFS, while it really effects the performance of SQUASHFS.

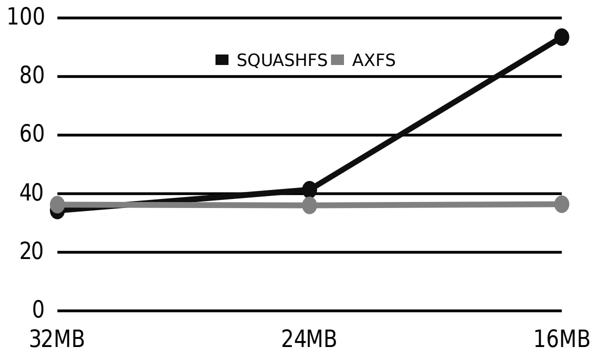

Figure 4: Impact of memory pressure on boot time (s)

5.3 Size

Comparing the image sizes for each filesystem, as we do in Figure 5, shows how AXFS compresses better than all the others, although it isn't much smaller than SQUASHFS. It also shows how optimized XIP AXFS is much smaller than the best we could do with XIP CRAMFS; both save the same amount of RAM.

Comparing the size of the XIP AXFS to the SQUASHFS image in Figure 5 makes it look as though the SQUASHFS is smaller. That is only part of the equation. If we take the amount of RAM used into consideration, it is clear an XIP AXFS image uses less total

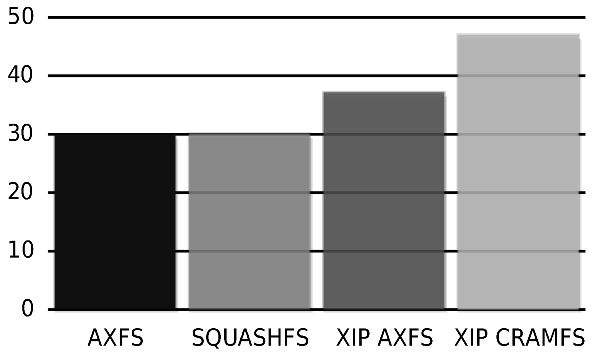

Figure 5: Image sizes (MB)

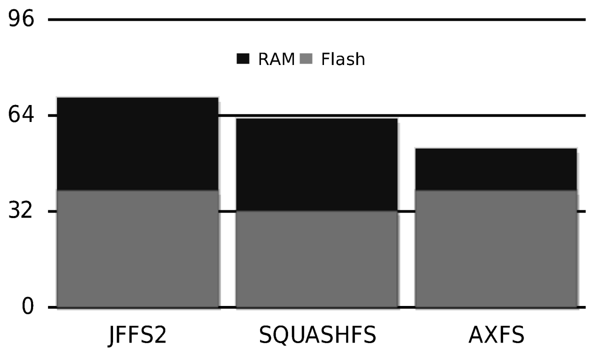

Figure 6: total used memory (MB)

memory, as shown in Figure 6. As all of these system use more than 32MB of Flash, a 64MB Flash chip would likely have to be used for all. However, while the JFFS2 and SQUASHFS systems both need 32MB of RAM, the AXFS system would only need 16MB. In fact, the AXFS system would still have more free RAM than the others, even with a smaller RAM chip.

6 Summary

AXFS was designed to create a filesystem with a chance of being merged into the kernel that provided the XIP functionality long used in the embedded world. It can conserve memory, saving power and cost. In the right circumstances it can make systems quicker and more responsive. Hopefully it can soon be merged and enjoyed by all—especially by those that have long been struggling with the XIP CRAMFS patches. We are also hopeful that other users will find new uses for its unique capabilities, such as the root filesystem in a LiveCD.

Proceedings of the Linux Symposium

Volume One

July 23rd–26th, 2008
Ottawa, Ontario
Canada