The Apparatus and Enabling of a Code Balanced System

Tony Benavides, Justin Treon, and Weide Chang
Flash Memory Group, Intel Corporation
California State University, Sacramento
Department of Computer Science
tony.benavides@ieee.org, justin.a.treon@intel.com, changw@ecs.csus.edu

Abstract

Success in the embedded world revolves around two key concepts: cost effectiveness and performance. The ability for an operating system to boot quickly combined with speedy application usage at runtime is important with regards to consumer unit adoption. The most common memory sub-system setup in cellular phone architectures today is what is called an eXecute-In-Place architecture. This type of memory sub-system defines the execution of code and data directly from NOR flash memory. An additional memory architecture of choice is called a Store and Download architecture. This is a memory sub-system where the compressed code gets copied to RAM at boot time and executes out of the RAM. This paper explores the addition of a new memory usage model called a Code Balanced System. The result is a system that combines a small RAM memory requirement with a performance increase for improved targeted application and boot time execution.

1. Introduction

The topic of this paper is the design and analysis of a new memory usage architecture that puts emphasis on speeding up the boot time responsiveness, minimizing the Bill of Materials (BOM), and reducing cell phone power consumption. The BOM affects the unit cost, which is the catalyst for the pricing of mobile phones today. Reducing the BOM will give more profit to the manufacturer and pass the value on to the consumer. Boot time performance is also a driving factor that many mobile phone providers push to strive for.

The current eXecute-In- Place (XIP) memory model works excellent for code execution and data storage. In systems that exhibit a situation where some data cannot take advantage of the cache in order to meet a limited latency requirement, it will be of a great benefit to focus on this Code Balanced System architecture.

When using a pure Store and Download (SnD) model, the battery life decreases due to the fact that more volatile memory has to be continually refreshed. This results in greater current consumption during both active and standby device modes. This paper investigates how to architect a memory sub-system in a way that will allow for low RAM usage, fast boot time and great application responsiveness. Of particular interest, will be the investigation of a Code Balanced System with regards to cellular phone design.

2. Related Research

The choice of a memory system implementation depends on the cellular market segment strategy targeted. For phones that are 3G based it is expected that high end systems are employed. These high end systems are envisioned as having a high speed execution bus to handle the fast multi-media streaming interface. Likewise, for 2-2.5G systems a different memory strategy is employed due to its change in performance elements. There could be a slower bus interface because of the less intensive processor tasks needed. Many different companies are researching different methods to interface with their own working memory solution.

Research in different memory sub-systems span over hardware and software elements. The number of memory buses, processor cores, and cache types has a large influence on the architectural model used for the memory subsystem. One flaw that is prevalent in computer architectures is the fact that usually a system is designed for one typical usage model. That model which works initially in the system is not further optimized once proven to have acceptable performance. Determining what constitutes a typical application model can be a huge challenge, and, according Hennessy and Patterson [1], a classic fallacy of computer architecture. The authors contend that there is no such thing as a typical application. This realization has pushed processor design to follow the primary goal of achieving acceptable performance over a wide range of potential tasks rather than optimizing for any one particular task. This ideology has migrated itself into the embedded memory subsystem as well.

Throughput characteristics of memory subsystems depend upon how well a given application maps to the architecture, not on how well the architecture can adapt to

the problem. Knuth's analysis of processor intensive applications reveals that most applications spend the majority of their time in a small subset of all the instructions that comprise a given application. This leads to the 90-10 rule in which 90% percent of the execution time is spent in 10% of the program [1]. Knuth's analysis is also applied to the software execution strategy in the designated memory environment. Thus, in order to improve performance, the focus should be on the frequently executed subset of applications that incur the most processor time. By escalating this analysis across application use cases, common candidates for throughput improvement can be grouped into cases based on RAM consumption with a vision towards developing a hardware memory sub-system that can accommodate most of these use cases. Architectures with this ability can capitalize on the performance benefits of customized memory footprint analysis by tuning itself to the current system applications while maintaining the flexibility of a total memory solution.

2.1. Related Research

An XIP memory model consists of a system that has NOR flash and PSRAM/DRAM within it. The code is executed from the NOR flash memory. The simplicity of execution is the key to its efficient memory usage. The execution of code consists of many random reads throughout the memory. The random act of reading a NOR memory cell and its quick read speed is the basis for an XIP solution. NOR flash read speed initial accesses are ~90ns. This speed, coupled with the many random reads of flash memory equate for a good execution solution of code. Execution of system code is not sequential due to many different areas of the binary that need to be accessed in order to run in a system (ex. branch and jump instructions). Fast code execution is prevalent when you use the XIP architecture. While the NOR flash consists of code and data areas, there is also needed RAM components. RAM memory accounts for certain changing elements such as the stack, heap, and non-constant variables. As you can see, the code and read-only data are read out of the flash memory which leads to the lowest RAM density required of all the memory architectures. In a runtime environment, the RAM is primarily for application use. While the density of RAM required in a system is at a minimum, this also allows the benefit of saving power. It is important to note that as a power savings addition, NOR flash memory can be turned off completely when not in use while still holding its data contents. This can be in the form of a Deep Power Mode pin on the device.

A typical XIP image will copy "Read-Write" sections of memory into RAM at boot time. All "Read-Only" sections are kept in Flash and executed/read from. This not only includes code but also constant data elements. The remainder "Zero-Initialized" sections are used for applications and typically set to zero.

Figure 1 explains how an XIP memory system works. On the right side of the diagram there is the code and data which reside in flash memory. Along with that is a picture of the systems working set of RAM. In a standard cellular execution stream, the code and data are read out of flash memory into the memory controller of the processor. The usage of code storage and data in the flash memory are the definitions of an XIP environment.

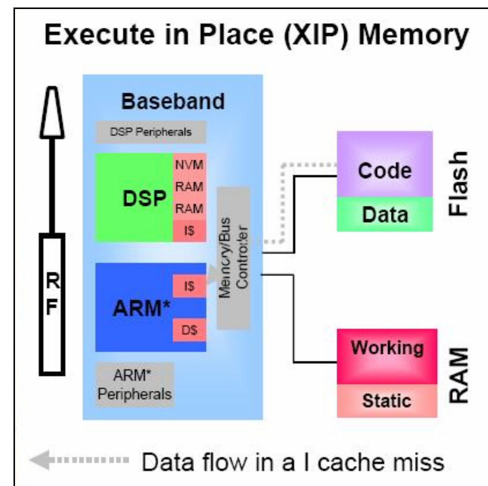

Figure 1. eXecute-In-Place diagram

Another popular architecture of choice is what is called a Store and Download (SnD) architecture. This is a memory system where the executable code is stored in Non-Volatile Memory (NVM) and copied to RAM at boot-up. At boot-up a small piece of NVM, usually within the micro-processor, runs code to copy the contents of the NVM memory (NAND or NOR) to the RAM and jumps to the RAM location. From that point the code and data are executed out of RAM. You can see that for a SnD system, large amounts of RAM are needed for the code execution. Since the code is compressed and stored in the NVM, a smaller memory density is needed to store the code in the NVM.

In the SnD model, the code is stored in the NVM as compressed data. When needed, the data will uncompress and get copied to the RAM. This creates more software overhead and complexity at boot time to copy all of the code over to the RAM. Extra RAM density is required to handle copying all of the code over to the memory. This extra RAM creates high standby power consumption due to RAM refreshing schemes. DRAM has to be continuously refreshed to retain its internal cell contents

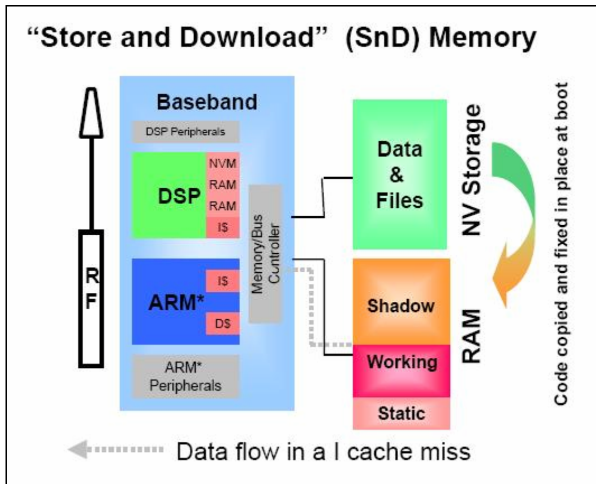

Figure 2. Store and download diagram

Demand Paged (DP) architectures can be defined as a subset of the SnD architecture. This is a method by which to minimize the RAM requirements at runtime. Code is stored compressed in NVM and will uncompress to the RAM when needed by the CPU. This is usually in the form of a page fault. Typically you will not have enough RAM in the system for all of the code. This method alleviates this issue by requesting that you have enough RAM for what is needed in the system at one time. Pages in virtual memory, not in physical memory, cause a page fault when accessed. In this method, the limited page pool is reused as different code pages are swapped in and out of main memory.

Figure 2 describes a working SnD model. In this configuration the NV storage copies over what is needed by the RAM at boot up. After this code is copied over then memory is mapped in/out as needed defined by the page fault handling scheme. The difference between this picture (SnD) and a Demand Paged environment is that with the DP you only use RAM that is needed at that time for code, along with other system RAM elements.

Performance aspects of DP are different that SnD. With the SnD methodology, all of the code is in RAM at once. This situation creates a faster access time than the DP method. The DP method is slower because of the time it takes to determine the need for a new page (page fault) and the time it takes for the software copy overhead (uncompression).

3. Code Balanced architecture

Embedded system designers have a wide variety of storage methods available for their intended projects. Depending on the hardware targeted and operating/file systems used, each has its own tradeoff between read speed, write speed, and compression. The key to generating a cost effective system is to balance the storage methods defined in the previous section. After analyzing the previous cellular memory architectures, the goal is to come up with solutions that will enable fast code execution speed while also speeding up the boot time and keeping the RAM density requirement low. The resulting investigation becomes a mixture between the DPmodel and the XIP model called a Code Balanced System. The reason this method is called Code Balanced is due to the process of taking each individual code element in the system and logically balancing its need to be uncompressed/XIP or compressed/DP depending on performance characteristics.

It is important to detect certain bottlenecks that occur in system memory access. By locating these hotspots to RAM you can remove the bottlenecks and reduce system latency. With this architecture, the code will remain in NOR flash while selected portions of applications and constant data segments get moved over to the RAM thus decreasing bus utilization. Most normal code for executing applications does not need to be moved to RAM because of the high Instruction Cache (I$) and Data Cache (D$) hit rate [8]. Since the I$ hit rates are high, the performance becomes the same as code execution when running from DRAM. With the understanding that certain object files can be manipulated to run compressed or uncompressed, you arrive at the basis for which the method of Code Balancing works. By targeting in on each object file you can investigate what will help system performance by putting the object in RAM (ex. data that does not keep up with D$ latency requirements), or keeping the executable in NOR flash to be run as code.

This Code Balanced interface allows the system engineer to optimize the XIP model previously discussed by being selective on what gets put in RAM and what stays uncompressed in NOR flash memory. To implement this takes knowledge of the platform hardware and software to make it work. The process takes place from a software standpoint by interfacing with the file system through a build process, generated map file, and linker scripts.

By analyzing the memory footprint of a XIP system it becomes clear that this method provides the most use of resources which in certain scenarios can give the best

value . Figure 3 describes a comparison of the amount of memory used in a system for either the DP or XIP solution. As you see, the XIP system takes less RAM than its counterpart. This allows the system designer to take advantage of the resources in the system instead of letting the memory remain unused.

Relating to the 90/10 rule, by recognizing the effect of being able to choose what in the system can be compressed or uncompressed you see the power of the Code Balanced architecture. Through system analysis, seldom used objects or libraries can be targeted to be saved as compressed code in NOR flash memory thus saving unused RAM space.

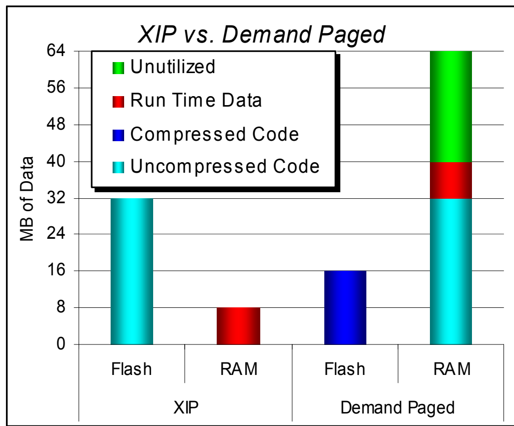

Figure 3. Memory footprint analysis

3.1. Architecture benefits

As Amdahl's Law states, to optimize on performance requires speeding up the time where the greatest latency is. By analyzing the hit rate and size of certain code segments you can also speed up performance, specifically during boot time. Migrating some of the constant data to high-speed RAM allows faster throughput for this data.

NOR Flash and the Code Balanced method provide the best savings of execution speed and standby power consumption in cellular platforms versus a DP model. The standby power consumption is largely due to the RAM array size. By reducing the size of the RAM array the refresh current is reduced, therefore increasing the battery life. By analyzing a typical handset usage model it can be noted that the percentage of the time is used talking on the phone is quite small compared to the percentage of time the phone remains in standby or idle mode. This is the time when the RAM will take up most of the system power.

Code executes more efficiently in a XIP system vs. a DP system because the cache makes up for any additional read discrepancy. Localized code is fed into the cache line and executed. Also, the additional software hit to uncompress the compressed code stored in the NVM adds more latency that the XIP system does not have.

Figure 4 relates how the Code Balancing system works. By selecting the pages from the Demand Paged situation and compressing them save memory space. The newly compressed files are then stored in flash memory. If you choose files that are not used that often (ex. printf, certain libraries, etc.) you will save on RAM utilization. You will take a uncompression hit when wanting to use these uncommonly used files but due to the codes hit ratio this will not be bad. You can see from the figure that you save on system RAM.

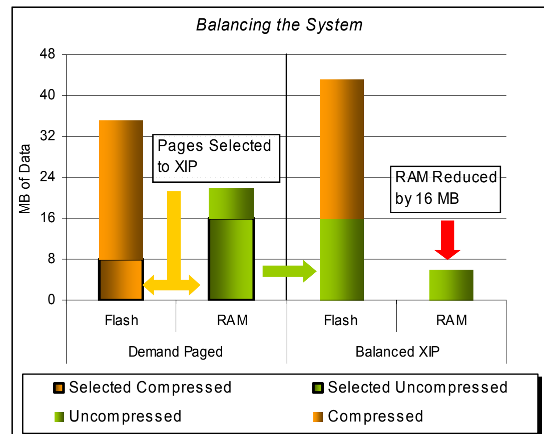

Figure 4. Analysis of balanced components

Within any memory sub-system with a compressed filing system component there is a "tug-of-war" between compression and the RAM usage. As compression of the filing system increases to reduce the size of the code stored in the flash memory, the system's RAM requirement increases to run the decompressed code. Alternatively, when running a portion of the filing system as uncompressed the RAM requirement decreases.

Uncompressed applications can be executed directly from flash without having to be buffered in the RAM. By adjusting the compressed to uncompressed code ratio, developers can purchase a smaller RAM/Flash chip combination while still achieving their performance goals. By achieving a Code Balanced system, developers will be able to save money, increase performance, and improve end user experience.

System engineers should primarily use a Code Balanced methodology as a tool to reduce the BOM by using the smallest and most cost effective RAM/Flash combination available. Focusing on the ideal system combination will add value by increasing application performance and boot time. By using a Code Balanced implementation for application code, the RAM density requirement is drastically reduced.

4. Tested configurations

The Linux Operating system was used to create the test results outlined. The system platform used in the experiments is an Intel PXA27x Development System. This is an embedded platform that consists of an Intel PXA270 Xscale processor, 64MB RAM, and 64MB M18 Intel Flash. For the Demand Paged version of data, a 64MB Small Page NAND device is used as the NV Storage element. The PXA270 has a 32kB internal Instruction and Data caches with each cache line consisting of 32bytes. Standard Linux build techniques were employed in this experiment. The JFFS2 and Linear CRAMFS were used along with the 2.6.14 based kernel.

The test results below consists of two configurations compared (Demand Paged vs. Code Balanced). For the Demand Paged scenario a NAND NVM was used with the JFFS2 and RAM. The Code Balanced solution consisted of a NOR NVM with RAM using a combination of Linear CRAMFS for Read-Write data and JFFS2 for Read-Only data files.

Figures 3 and 6 refer to the data collected on the PXA27x Development Platform. Figures 4 and 5 refer to data collected on a separate PXA270 development board with 96MB of NOR flash and 64MB of RAM.

4.1. Test results

The Code Balanced model provided the ability to decrease boot time and minimize the total memory package cost. Figure 4 outlines how the Code Balanced solution efficiently utilizes memory space. It is easily seen that this new method uses less space than its Demand Paged counterpart. This is due to the fact that by balancing the object components the unutilized RAM code is compressed and put into NOR flash memory. The flash portion of XIP will increase because you are putting more uncompressed data in the flash. The RAM decreases because a great portion of the system code and data are now in flash.

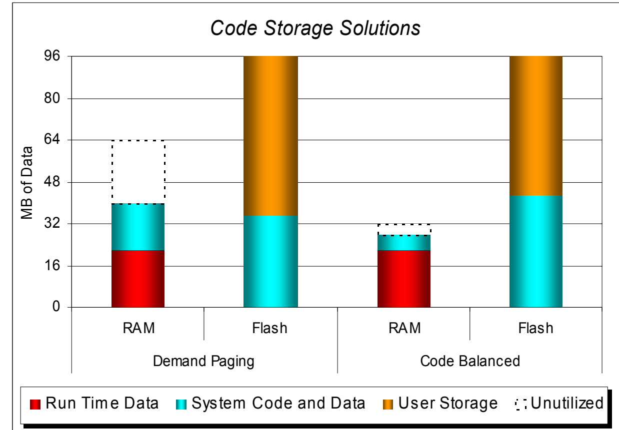

Figure 5. Memory usage comparison

The boot time performance also increases when the Code Balanced method is compared to the DP. The reason that the Code Balanced method is faster to boot stems from the process that the system follows at reset. After the reset is triggered, the processor executes code at address 0x0. For a Code Balanced system this means that the code is executed immediately after reset.

Comparatively, the DP system boot loader must first load the kernel into RAM and decompress it before control can be passed to the kernel initialization routines. Moving the kernel and decompressing it is not required on the Code Balanced system. As noted in Table 1, this helps the boot time by a factor of 2.2x.

Table 1. Benchmark comparisons

Technique	Boot	Quake	Browser
Code Balanced	32.2	20.0	2.6
Demand Paged	71.6	26.7	3.9
DP Slower by	2.2X	1.3X	1.5X

When analyzing how applications launch in a Code Balanced architecture, it can be seen that time to launch the GUI was reduced. This is because the uncompressed/XIP pages do not need to be buffered into RAM. The DP based system is primarily slower due to the demand paging overhead and page faults.

Also investigated was the launch of the application Quake and a browser used in Linux. These results, although not as apparent as the boot time numbers, helps to solidify the effects of a Code Balanced environment. The launch time for the first person shooter Quake was 1.3x slower on the strictly DP system versus the Code Balanced implementation. When launching the Konqueror Web Browser the DP implementation was

1.5x slower to launch versus the Code Balanced implementation.

By analyzing the differences between Demand Paged vs. Code Balanced system it is clear to see where the reduced BOM comes from. As the engineer selectively compresses object code, she/he can balance the needs of the system by allocating what needs to be compressed and what can stay in NOR Flash memory. For example, in the Demand Paged situation there is a lot of RAM used during run time. The ability to find out what objects are being used the least and then assigning some of them to flash memory allows the RAM utilization to be increased.

5. NAND Flash Memory

For years NOR flash has been the major player with regards to non-volatile memory in portable electronic devices. NAND flash's main focus group was targeted primarily as a large-density storage medium for use in consumer devices. These devices range from solid-state storage disk drives to digital camera memory storage.

Within the past year the NAND flash memory has started to become a clear solution in areas where before, only NOR was the complete winner. Memory controllers that were once NOR-centric are now coming out with built in NAND controllers that enable the booting of NAND and the ability to handle the error correction in hardware. This is allowing NAND to enter into consumer devices where before they were only used sparingly.

Cellular phone devices are starting to focus on providing customers with data driven interfaces including MP3 players and video viewing/streaming. This is driving up the total phone density in a cellular phone. As a result, the designers of memory sub-systems are opting to design in NAND solutions where traditionally NOR was only used.

In a cellular phone, since NAND cannot be used to access random data, the best way to utilize NAND is to use the device in a Store and Download methodology. Refer to Figure 2. In this memory architecture, the NAND will be used along with SDRAM or DDR in the embedded system. With the data usage increasing per embedded device, this could be a viable solution in the future and would exclude both NAND and NOR in a system.

The NAND Store and Download model would work as explained in Section 2. The processor would load code from the NAND to DRAM at bootup and run the code from the DRAM. The NAND device would be used as data storage in is non-volatile cells. This architecture is a little more complex than its NOR eXecute-in-Place counterpart due to the fact that you also need a special NAND controller to be able to boot from the device to get the code copied over to DRAM. There is also controversy in this architecture because some people believe that the bus between the NAND and DRAM could be snooped which could pose a security risk to the code contents. The industry is moving to MCP (Multi-Chip Package) which would house both the NAND, DRAM and micro-Processor in a same package which could alleviate this risk.

Although the initial engineering and floor costs are more involved, the end result could be a less expensive unit. This is because you design out the more expensive NOR memory and cater to only the NAND and DRAM in the system. The negative to using NAND in this type of model is that there is some extra error correction that needs to happen when using NAND memories and this correcting action can take time. The ECC can manifest itself in either a hardware or software form. The hardware form, although additional, proves to be better because of its ability to act quicker than its software counterpart. In addition, using a Store and Download model tends to require more power as the DRAM tends to take more of an active role.

While the question continues as whether to go with a NOR or NAND memory architecture solution, there is no doubt that many of the cellular providers are looking at both solutions. The question still revolves around cost, speed, and ease of implementation.

5. Conclusion and future work

A Code Balanced methodology is not against the use of NAND and RAM in a system. This memory architecture encourages analyzing the software system components so that a better judgment can be made on where to put files and applications. This puts the system architects more in the driver seat to make decisions that will affect the cell phone bottom line and pass good quality, high responsive, low cost phones to consumers.

Customers will perceive a system as being slow if the boot time or application launch time is slow. Boot time along with a minimized RAM/Flash memory package are the most notable difference between a DP and Code Balanced system.

Discussing the exploration of using the NAND flash memory as a possible alternative to NOR, it would be good to see more investigations catered at comparing the two. This could consist of both performance measuring, including Operating System overhead and filesystem manageability.

For future investigations it would be interesting to conduct this same experiment for both a WinCE Embedded system and a Real Time Operating System. The analysis of the boot time and application performance parameters for these OS's will complete this investigation.

This includes comparing the NAND Store and Download model with the Code Balanced model to explore what can happen to increase performance and relieve the complexity of adding a NAND system solution.

6. References

[1] J. Hennessy and D. Patterson, Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers, Inc., 1996

[2] Linux MTD, http://www.infradead.org, 2006

[3] Linux Open Source Kernel, http://www.kernel.org, 2006

[4] Steven J. Vaughan-Nichols, OS's Battle in the Smart-Phone Market, 2003

[5] Cliff Brake, Jeff Sutherland Flash File systems for Embedded Linux, http://www.linuxjournal.com/node/4678/print, 2003

[6] MontaVista, XIP Linux Execution, http://tree.celinuxforum.org/pipermail/celinux-dev/2004-November/000220.html, 2004

[7] Howard Cheng, Eric Anderson, Mike Edgington Optimized XIP Collateral, Intel FMG Ecosystem Team, 2006

[8] Jared Hulbert, Justin Treon, Linux Optimized XIP Collateral, Intel FMG System Analysis Team, 2006

[9] Weide Chang, George Landis, GP-DISP: A general purpose dynamic instruction set processor, 2004

[10] Michael Santarini, NAND versus NOR, 2005