Dynamic Memory Placement of Applications on Extreme Resource Constrained Hardware

David Scheidt

February 4, 2020

Abstract

In recent years Non-volatile memory (NVM) technologies have evolved to an energy efficient alternative for main memory technologies, especially in extreme resource-constrained devices. Such devices benefit from NVMs lower power consumption, therefore allowing to build devices for novel fields. However, NVM technologies show a drawback in their lower write endurance compared to traditional technologies. While the locality of references enables optimization techniques for traditional memory types, a write-intensive application can quickly wear out individual cells of a non-volatile memory leading to a decrease in lifetime. Both hardware- and software-only solutions have been proposed to address this issue, by spreading the memory access across the entirety of the memory in a process called wear-leveling. By applying such techniques the lifetime can be drastically improved.

This paper explores a software-only wear-leveling technique and exposes the Executable and Linkable Format (ELF) to improve the approaches ability to dynamically place an application in the memory for execution. The ability to relocate an application promises to increase the memory lifetime, by allowing the wear-leveling algorithm to make write accesses more uniform across the memory. Especially in extreme resource-constrained device, where little to no memory hardware is available, the proposed solution helps to increase the memory lifetime.

1 Introduction

Utilizing NVM technologies as the main memory of a computer comes with several advantages over Dynamic Random-Access Memory (DRAM). With their lower power consumption and higher integration density, they allow to build smaller, cheaper and more energy-efficient devices. Applications being applied in environments with a lot of radiation, such as space, may also benefit from their high tolerance against effects caused by radiation [1].

On the other hand side, NVM technologies show a lower write endurance per memory cell compared to traditional memory. Figure 1 illustrates that a Resitive Random-Access Memory (ReRAM) cell, having the highest of all shown NVM technologies, still has four orders of magnitude less write cycle endurance than DRAM. In order to compensate the resulting decrease in lifetime the process of wear-leveling can be applied. Wear-leveling tries to balance the wear on individual cells across the entirety of the memory, leading to a more uniform wear of all cells. This usually requires specialized hardware e.g. to access the write count of a cell. As this work focuses on extreme resource-constrained hardware, software-only approaches are considered, since additional hardware might not be available. Such software-only approaches can utilize virtual memory to achieve wear-leveling on a per-page level, which does require the widely-available Memory Management Unit (MMU). In the absence of e.g. an MMU, software-only approaches usually cannot deliver any improvement of the lifetime of NVM cells.

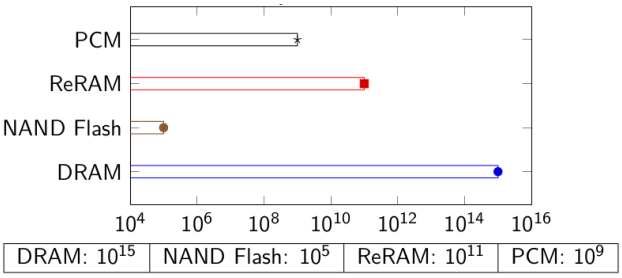

Figure 1: Write Cycles Endured by Single Cells of Selected NVM Technologies [2]

This work proposes a tool to allow placing applications anywhere in the memory, which in turn enables wear-leveling even when a MMU is not available. While traditional approaches use a MMU to map virtual memory to physical memory, this tool allows to physically place an application at any base address. This way a similar effect to the traditional approach can be achieved. In addition, the tool can be used in conjunction with wear-leveling through virtual memory, in order to reduce non-uniform write accesses inside a memory page.

2 Related Work

In previous work [4], a full system simulation based on C/C++ is used to evaluate a novel approach to in-memory software-only wear-leveling. Using the gem5

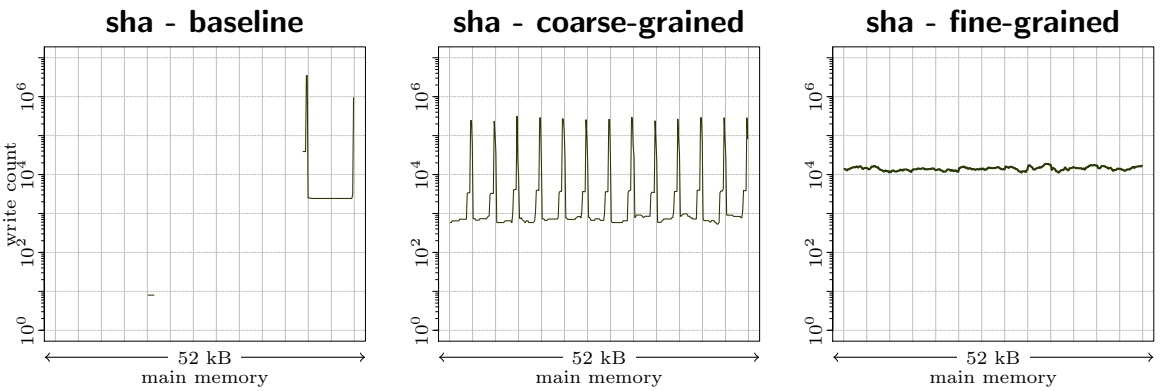

Figure 2: Write Count Distribution of a Secure Hash Algorithms (SHA) Calculation Applying Different Wear-Leveling Approaches [4]

simulator, a 64bit ARMv8 Cortex A53 running a custom bare-metal runtime system, is first utilized to create baseline benchmarks of several applications, such as the calculation of a SHA sum.

Figure 2 shows the results when applying no (baseline, left) wear-leveling, wear-leveling based on virtual memory (coarse-grained, middle) and the novel approach (fine-grained, right). The latter reloactes the stack of the application on a regular basis in addition to the wear-levling done on the memory page level. This yields an overall more even result and improve in lifetime compared to no wear-leveling of a factor of 900 [4]. However, it is also found that the achieved improvement differs strongly depending on the application, because only the stack of the application is moved through the memory. The lifetime therefore suffers from applications, that do not access their stack a lot, but tend to write into other sections of the applications memory. In traditional application memory layouts this especially applies to the *data* and *bss* as these hold global and static data. The ability to place the entire memory layout of an application anywhere in the physical memory, allows to overcome this issue.

3 Preliminary Work

For this work the simulation setup including the bare metal runtime system from [3] is adopted and adjusted to prepare it for placing applications dynamically.

Originally the application is compiled as a part of the runtime system and thus is placed in the same binary as the runtime system. This predefines the location of the *data* and *bss* segments inside the runtime systems memory space. This dependency is first removed by extracting the application from the build process of the runtime system.

In a second step, in order to use debug outputs from the application and ease future development, the runtime systems log mechanism is changed to make use of system calls instead of function calls. This way both the runtime system itself and the application may use the log mechanism using a simple assembly instruction.

3.1 Runtime Application Separtion

The first step to separate the application from the runtime system is to move the source files to their own file structure and removing all references to those files from the runtime system. That way, when launched, the runtime system does all of its initialization and then idles. This is later replaced by loading the elf file (cp. section 5) to restore the systems original functionality.

Afterwards the application is built using the same toolchain as used for the runtime system. As the runtime system lacks a file system, there is no way of loading the resulting binary from a file. To resolve this issue, the binary is converted to a source file containing a byte array that represents the binary. This source file is then copied back to the runtime systems source files, allowing to access the binary the same way it would be accessed through a filesystem.

3.2 Interrupt-based Debug Log

Previously the runtime system provides a simple text output by implementing a device driver for the Universal Asynchronous Receiver Transmitter (UART) controller of the simulated hardware. The output of the resulting UART console is written to a file by gem5, which can be monitored during runtime. In addition, access to the driver is wrapped in a singleton-type class supporting text formatting, as the UART controller only allows to output a single character. This eases writing outputs such as pointers and floats, but also leads to a dependency between the wrapper class and the driver.

For separating the application from the runtime system, access to the driver from the wrapper class is replaced by an ARM supervisor call using `65535` as the immediate argument [7, p. 1325, C6.2.316]. The runtime systems interrupt routine handler then expects the output string and its length in the general purpose registers `X0` and `X1`. The interrupt handler then utilizes the driver to output the string and returns execution.

As the wrapper class does not have any additional dependencies, it can be compiled together with the application in a "standard library"-type way. With this change, both the applications memory space and

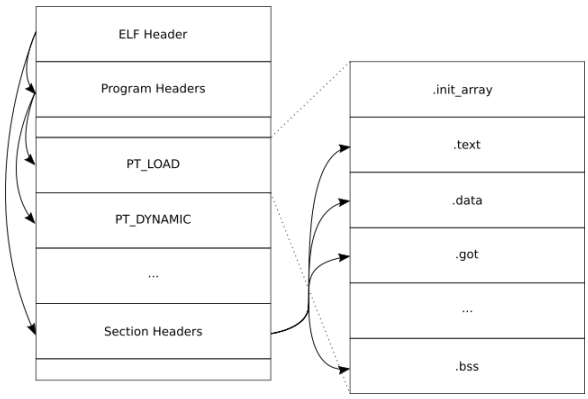

Figure 3: Example of an ELF File in Execution View

the runtimes memory space contain their own singleton instance. This way, the application does not need to access any of the runtime systems memory space directly. Since the runtime system is running with elevated rights (Exception Level 1), memory access through the application directly would yield an exception. Using its own singleton instance of the logger, the application is therefore presented with a safe way of interacting with the driver.

4 Executable and Linkable Format (ELF)

The ELF specification is first published in 1993 and got an update to its most recent version in 1995 [5]. It can be found in the Tool Interface Standard v1.2 [6]. ELF is a widely used format for executable files and is part of the Application Binary Interface (ABI), which is designed to standardize interfaces used by developers across multiple operatings systems. Over the recent years, ELF is quietly adopted as the go-to standard, as many Linux-based operating systems, such as RedHat, Debian and Android use it. The following paragraphs briefly summarize the specification from the Tool Interface Standard v1.2 [6].

Generally ELF provides two views to a binary file: The linking view and the execution view. While the former one is used when building a program, the latter one is used for running the program. As the scope of this work is on loading an already built program into memory, the execution view is outlined in the following.

Figure 3 shows the structure of an exemplary ELF in execution view. Every ELF starts with a 52 or 64 byte long header for the 32bit or 64bit respectively. The header acts as an index for the structure of the rest of the file. The arrows in figure 3 indicate references from the header to different parts of the file. Noteable fields in the header are the `e_entry` (memory address of the first instruction of the program to be executed), the `e_type` (identifier for the type of file e.g. library, executable, etc.) and the `e_phoff` and `e_shoff` (point to the program headers and section headers).

The header is usually followed by the program headers, which is a table that is used to create an image of the file, that can be run by the system. It therefore specifies if a so-called segment, e.g., needs to be loaded into memory, at which virtual memory address the segment should be placed at and if additional linking to dynamically linked libraries is necessary. When given an ELF file, the loader has to copy the specified segments into memory. If the ELF references shared libraries provided e.g. by the operating system, the dynamic linking process needs to be invoked. This process resolves references during loading of the file, which can be found in specifically designed sections. All sections of the file are referenced by the section headers, which is a table that lays out the position of, among others, the text, data and bss sections inside the segments.

4.1 Position Independent Code

ELF supports, besides absolute code, position-independent code of which there are two types: Position-independent Code (PIC) and Position-independent Executable (PIE). PIC is designed to be used for building shared libraries, while PIE utilizes some of the functionality of PIC to provide position-independent executables. Both types utilize relative addressing and must be set during compilation of the executable. They allow to alter the logical address of the process arbitrarily as long as the relative position of the processes segments is kept.

In the following the focus is set to PIE ELF files. While adresses in a non-PIE follow the conventions set for virtual memory adresses (e.g. it is common to place executables at `0x400000` in virtual memory), adresses in a PIE are relative to the beginning of the respective ELF file. Since all instructions use relative addressing, this does not conflict with the execution of those instructions. However, global and static objects, which are being placed in the data section, cannot be accessed in this way as their address is unknown during compilation. To overcome this issue a Global Offset Table (GOT) is used in PIE. The GOT introduces an additional level of indirect memory access. Whenever a global or static object is accessed, instead the GOT, whose address is known during compilation, is used. This implies that the GOT has to be updated during loading of a PIE ELF to point to the correct global or static objects.

5 ELF Loader Implementation

The goal of the proposed ELF loader is to allow the runtime system explained in section 2 to load an ELF to any position in the memory. Achieving this, the runtime system may utilize its present logic for write-count estimation and relocation to further increase the lifetime of individual NVM cells, particularly for applications that heavily utilize the data section.

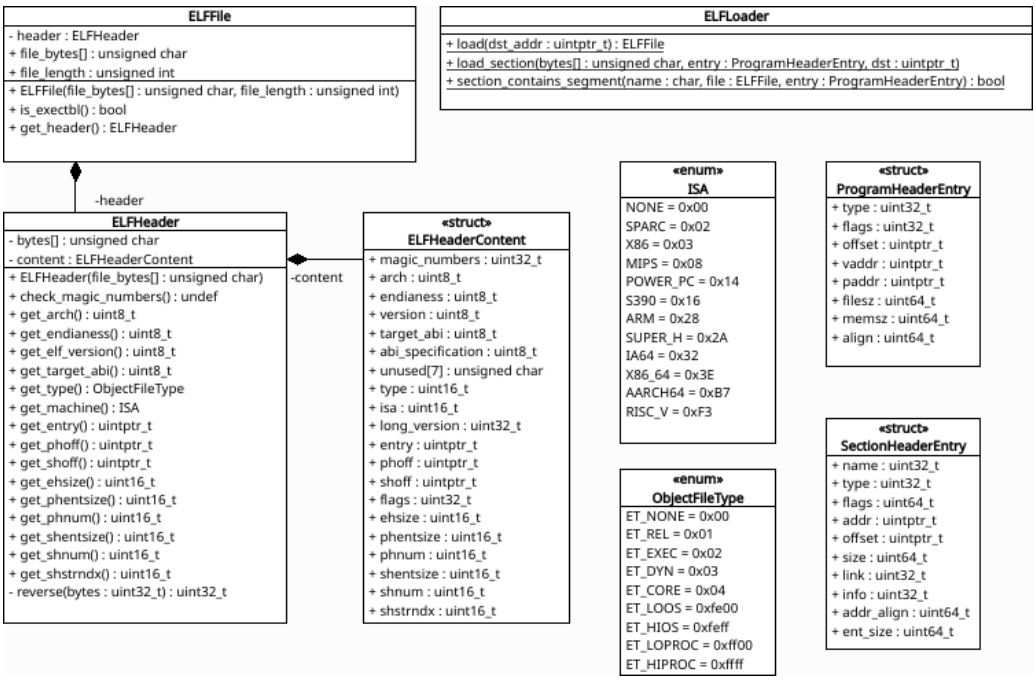

Figure 4: Class Diagram of the ELF Loader

The loader is implemented using an object-oriented approach following the class diagram shown in figure 4. The program flow of the main `ELFLoader` classes `load` method is set as follows:

1. Parse and check the ELF
2. Copy loadable segments to the desired address given by runtime system
3. Update the GOT (as discussed in 4.1)
4. Initialize static and global variables
5. Calculate entry point address and handover to application

Details to the individual steps are discussed in the following.

5.1 Parsing and Checking an ELF

As discussed in section 3.1, the runtime system lacks a file system, hence the binary ELF is converted to a source file that is being compiled with the runtime system. In order to make the loader implementation independent of this fact, it requires the runtime system to hand over a byte array and its length. This can be easily achieved from the source file as it already contains this information or from a filesystem that might be added to the runtime system.

The first step in checking the ELF file is to check the first four bytes of the file for the magic number (`0x7F 0x45 0x4c 0x46` / `'DEL' E L F` as ASCII characters).

If the magic numbers match, the loader creates and object-oriented representation of the ELF file. The ELF header, as well as the program headers and section headers are parsed and objects are created from them. These objects hold internal data structures that map to the memory and allow to access individual fields of the headers through their corresponding `get` method (e.g. a `ELFHeader` object exposes the entry point address through its `get_entry()` method).

When the file is loaded, some assumptions are made due to the choice of the runtime system: The architecture is 64bit (AArch64, determined using the `ISA` struct and the `get_arch()` method) and the file is executable (determined using `is_exectbl()` method). Furthermore a statically linked PIE is assumed to enable dynamic placement of the image. If these conditions do not hold, the loader aborts further processing the file and causes the runtime system to stop.

5.2 Copying Loadable Segment

Copying the loadable segments of the ELF file is required to create a runnable image of the application. To achieve that, the `ELFLoader` utilizes the `ELFHeaders` `get_phoff()` ('Program Header Offset') method. This allows the loader to find the `ProgramHeaderEntry` structs in the byte array representing the ELF file. If the type of a program header entry is `0x00000001`, it defined is loadable (PT_LOAD) and copied to its destination address. The destination address can be calculated from the destination base address given to the loader in `load(dst_addr: uintptr_t) : ELFFile` and the `ProgramHeaderEntry`'s `vaddr` property.

The actual loading of a segment is implemented in `load_section(bytes[]: unsigned char, entry: ProgramHeaderEntry, dst: uintptr_t)`. Whenever a segment is loaded using this method, the `filesz` and `memsz` properties of the respective program header entry has to be respected. These two properties specify the size of the segment in the ELF file (`filesz`) and in the image in memory (`memsz`). In general `memsz` is

larger than or equal to `filesz`. This is due to the fact that e.g. the segment may contain the bss section, which does not take up any space in the ELF file but still needs to be allocated to run the program. The loader therefore sets the memory between `filesz` and `memsz` bytes at the destination memory space to zero, whenever it loads such a segment.

5.3 Updating the GOT

The GOT is a section inside one of the loadable segments of a ELF file. In order to update the entries of the GOT the loader has to perform two actions: First it needs to find the GOT section in the ELF file and identify its offset relative to the beginning of the file. Next the loader can use that information to find the offset of the GOT in the target memory space of the application and go through its entries.

Once the above steps are performed the GOT entries can be updated by adding the loadable segments base address to the relative address of the variable already present in the GOT, resulting in the absolute address of the variable in memory.

5.4 Initializing Static and Global Variables

In typical C/C++ environments the runtime library of the respective language handles initialization of static and global variables in the runtimes startup files. This includes setting up preliminary requirements (such as finding the `argc` and `argv` arguments) and calling the applications `int main(int argc, char argv[])` function. However, since the loader is run in a bare metal runtime system, the runtime libraries are missing and the loader needs to take over the aforementioned responsibilities. Per convention the applications run from the ELF do not utilize any arguments for their `main()` function, hence the loader only has to take care of static and global variable initialization.

For that purpose ELF holds a section called `init_array`. The init array contains functions that need to be called in order of appearance in the file to e.g. call constructors of static or global objects and to initialize any variables. The loader first finds the section similar to the GOT and then executes the referenced functions using function pointers. This effectively executes parts of the application with elevated rights (the loader is run from exception level 1) and is done for simplification of the process.

5.5 Entry Point and Handover

The entry point of the application is specified by the `ELFHeaders` `get_entry()` function. Similar to the process described in the previous sections, the actual address of the entry point is hence calculated by the value returned from that function and the destination memory space address. The actual handover to the application is done by the runtime system, whose startup code is adjusted to calculate the entry address and setting it as the return address for exceptions returning from exception level 1, on which the loader is run. The final exception return statement of the startup code then hands over the control to the application running on exception level 0.

5.6 Loading Segments Individually

During the course of this work, options to load the segments of a program individually from each other are explored. If the ability of the loader can be further enhanced to be able to e.g. move only the data segment through the memory, the wear-leveling approach could be applied in a more fine-grained manner. PIE ELFs are found to be a promising technique for achieving this.

However, as of the time of writing, basic static variables (such as `int`) in PIE ELFs are still referenced directly using relative addressing, instead of utilizing the GOT. This necessitates to keep the offset constant, between the `text` and `data` segment.

The `-mno-pic-data-is-text-relative` compiler flag may help to resolve this issue, but is not (yet) available for the toolchain (aarch64-linux-gnu-g++) used for the project. The flag leads the compiler to assume that the offset between `text` and `data` segment is dynamic. Therefore, the offset is unknown during linking of the application, which prevents relative addressing in the `text` segment, to access known data in the `data` segment. The flag further implies that a specific register of the cpu is used to store the address to the GOT. Adding support for this feature to the loader would hence require to update the register in addition to the GOT itself, whenever an ELF is loaded into memory or the `data` segment is moved.

Note that this option references PIC type ELF files. Since PIE is a subset of PIC, these options, once available, should also have effect on PIE type ELF files.

6 Conclusion

The ELF loader proposed in this work allows to utilize statically linked PIE ELF files to load applications to any valid memory position (alignment has to be respected). This implementation eliminates the need for a MMU to perform wear-leveling using the approach proposed in [3] and should allow to increase the found lifetime improvements of individual cells further by applying the concepts to the data/bss section of the application. The concept of position-independent code, specifically PIE binaries, is exposed to achieve dynamic placability of the application with commonly available compilation tools.

The runtime system used as a starting point for this work is further improved by refactoring the contained logging mechanism to use system calls instead of using the driver directly. This allows the logger to be statically linked into applications built for the runtime system in order to enhance the debug experience and increase verbosity of the application during runtime of

long simulations. The loader is further tested under exemplary conditions, in order to show that it is operational and to yield predictable setups.

For future work the impact of dynamically placing the individual segments in the memory on the lifetime of the NVM cells has to be investigated using multiple benchmark applications with different levels of usage of e.g. the data/ bss section.

References

[1] Irfan Fetahovic, E.Ć Dolićanin, D.R. Lazarević, B.B. Lončar, *Overview of radiation effects on emerging non-volatile memory technologies*, Nuclear Technology and Radiation Protection, 2017

[2] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao, *Emerging nvm: A survey on architectural integration and research challenges*, ACM Trans. Des. Autom. Electron. Syst., November 2017

[3] Christian Hakert, *Memory Access Analysis and Endurance Leveling Approaches for Non-volatile Working Memory Systems*, online, `https://ls12-joomla-host.cs.tu-dortmund.de/daes/media/documents/theses/2019-hakert.pdf`, Last Accessed 06.01.2020, Published 2019

[4] Christian Hakert, Kuan-Hsun Chen, Mikail Yayla, Georg von der Brüggen, Sebastian Bloemeke, Jian-Jia Chen, *Software-Based Memory Analysis Environments for In-Memory Wear-Leveling*, 25th Asia and South Pacific Design Automation Conference ASP-DAC 2020, Invited Paper, 2019

[5] Linux Foundation, *Referenced Specifications*, online, `https://refspecs.linuxfoundation.org/`, Last Accessed 02.02.2020

[6] TIS Committee, *Tool Interface Standard (TIS)*, online, `http://refspecs.linuxbase.org/elf/elf.pdf`, Last Accessed 02.02.2020, Published May 1995

[7] ARM Limited, *ARM Architecture Reference Manual*, online, `https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile`, Last Accessed 28.01.2020