Linux Kernel Compaction through Cold Code Swapping

Dominique Chanet[1], Javier Cabezas[2], Enric Morancho[2], Nacho Navarro[2], Koen De Bosschere[1]

[1] Department of Electronics and Information Systems
Ghent University
B-9000 Ghent, Belgium
{dchanet,kdb}@elis.UGent.be
[2] Department of Computer Architecture
Technical University of Catalonia
E-08034 Barcelona, Spain
{jcabezas,enricm,nacho}@ac.upc.edu

Abstract. There is a growing trend to use general-purpose operating systems like Linux in embedded systems. Previous research focused on using compaction and specialization techniques to adapt a general-purpose OS to the memory-constrained environment presented by most embedded systems. However, there is still room for improvement: it has been shown that even after application of the aforementioned techniques more than 50% of the kernel code remains unexecuted under normal system operation. We introduce a new technique that reduces the Linux kernel code memory footprint through on-demand code loading of infrequently executed code, for systems that support virtual memory. In this paper, we describe our general approach, and we study code placement algorithms to minimize the performance impact of the code loading. A code size reduction of 68% is achieved, with a 2.2% execution speedup of the system-mode execution time, for a case study based on the MediaBench II benchmark suite.

1 Introduction

In recent years, embedded systems have become increasingly complex. For example, mobile phones have evolved from relatively simple devices that provide phone calls and text messaging to veritable multi-media devices that also take pictures, play music and movies, surf the Internet and have extensive contact management and calendaring functionality. Due to this trend, the complexity of the software running on these devices has risen exponentially. Developers turn more and more to pre-built components and high-level programming languages in order to meet the functionality requirements and time-to-market pressure in highly competitive markets. This also concerns the operating system used on these devices: there is a growing trend to use general-purpose operating systems. Most of the attention has gone into Linux, as it is freely available, and its

open source nature gives developers full control, allowing them to adapt the OS in any way they see fit.

General-purpose operating systems offer a large number of functionalities that are unneeded in embedded devices, as they have been developed for desktop or server computers that must support a very wide range of applications and peripheral devices. The use of such systems can vary widely over their lifetime. Embedded systems, by contrast, usually have a well-defined and limited functionality, with software and hardware configurations that do not change over the device's lifetime. As embedded systems typically have strict memory constraints, it is desirable to remove as much of the overhead incurred by the unnecessary features of general-purpose operating systems as possible. Part of the overhead can be avoided by configuring the OS kernel appropriately at build time to exclude unnecessary drivers and features. However, this configuration facility is usually not fine-grained enough to remove all of the unneeded code and data. Recent research on link-time compaction and specialization of general-purpose OS kernels [7, 19] has shown that even on a fully-configured Linux kernel significant amounts of code can be removed if the hardware and software configuration of the target system are known and fixed.

However, in [8] it is shown that, even after application of the aforementioned compaction and specialization techniques, less than half the code in the kernel is executed during normal system operation. Part of the unexecuted code is there to handle unexpected situations, like hardware failures. The other unexecuted code is in effect unreachable, but is not detected by the aforementioned specialization techniques due to the limitations of static analysis. The authors of [8] propose to store all the unexecuted code (henceforth called *frozen code*, a term introduced by Citron et al. [9]) in memory in a compressed form, and later decompress only those parts that are needed at run time. However, under their approach it is impossible to determine a hard upper bound for the amount of memory the kernel's code memory footprint, as once-decompressed code cannot be removed from memory any more due to concurrency issues that arise from the inherent multithreadedness of the Linux kernel.

In this paper, we propose a novel approach to solve this problem for systems that have support for virtual memory. While virtual memory support is not yet available for all embedded devices, it is already supported by several important embedded processor families, such as the Intel XScale [2], the Texas Instrument OMAP [4], and the MIPS 4K [3]. Based on profile information, our technique selects code that will be put aside from the kernel's resident memory image and loaded on demand whenever it is needed. To avoid the high latencies of loading code from disk, the removed code will be stored in a fast off-line memory (e.g. Flash memory). Contrary to the aforementioned approach [8], our technique allows to determine an upper bound on the kernel's code memory footprint. In this paper, the technique is evaluated for the Linux kernel on the i386 architecture, but it is easily portable to other architectures and operating systems.

For this paper, we have chosen to focus on Flash memory as the secondary memory in which the not-loaded code is stored. We believe Flash memory is a

good fit for this technique, as it is sufficiently fast, and it is typically already available in embedded systems to store the device's firmware. Some typical devices that can be targeted by our technique are the Linksys WRT54GL wireless internet router (16 MiB of RAM, 4 MiB of Flash ROM), the Linksys NSLU2 network storage server for home networks (32 MiB of RAM, 8 MiB of Flash ROM) and the Devon NTA 6010A thin client (64 MiB of RAM, 64 MiB of Flash ROM). All of these devices have a fixed function, and all of them run the Linux kernel as their operating system.

It is important to note that the technique proposed in this paper is not the ultimate solution to the memory woes of embedded systems. Rather, it is a building block in a total solution. The OS kernel is only a part of the software that runs on the device. It is equally important to reduce the memory footprint of the user space programs through techniques such as link-time compaction [10] or code compression [11].

The remainder of this paper is organized as follows. In the next section, we give a general overview of our approach. Section 3 details the code selection and layout algorithms that are instrumental to limiting the performance impact of our technique. In Section 4 we discuss the implementation. In Section 5 the technique is evaluated. Section 6 reviews related work, and we draw conclusions and present future work in Section 7.

2 General approach

In the most general sense, we wish to develop an on-demand code loading scheme in order to reduce the static RAM footprint of an operating system kernel. The most important design criteria are:

- *Reliability*: the correct working of the kernel must in no way be compromised by the code loading scheme.
- *Performance*: the OS kernel is a performance-critical part of the system, especially as it communicates directly with the hardware devices. Loading code must not slow down the system too much.
- *Guaranteed size reduction*: unlike the approach proposed in [8], our approach should guarantee a hard upper bound on the kernel code memory usage, and it should be considerably lower than that of the original kernel. Note that this does not mean that the user can set a limit up front (e.g. "the kernel code should only use x bytes of memory") but rather that, after our technique is applied, the user knows exactly how much memory will be used by the kernel code.
- *Transparency*: there should be no major rewriting or manual annotation of the kernel code necessary.
- *Automation*: the code to be loaded on demand must be selected and partitioned into loadable fragments automatically. Users of the technique should not need to have an intimate knowledge of the kernel code.

The requirement of a hard upper bound on the kernel code's memory usage implies that any suitable scheme not only implements on-demand code loading,

but also code *eviction* whenever the memory allocated for the kernel code is full and new code needs to be loaded. However, due to the inherent multithreaded nature of most OS kernels, this raises concurrency issues. If a code fragment has to be evicted, the scheme must ensure the reliable execution of the kernel threads that may currently be executing the evicted code fragment. Automatically inserting locks in the kernel code to protect the loadable code fragments from untimely eviction creates a very real risk of introducing deadlocks. Therefore, an alternative to the introduction of locking must be found to guarantee the kernel's reliability.

In the remainder of this section, we first investigate the Linux kernel's module loading scheme, which can be considered a form of on-demand code loading, and then propose our own solution to the problem.

2.1 Linux Kernel Modules

The Linux build system allows the developer to compile parts of the kernel (e.g., certain hardware drivers) as loadable modules. These modules can then later be loaded on demand when their functionality is needed. This facility is mostly intended for the distribution of kernels for generic machines. Drivers for a wide range of peripherals are compiled as modules, and upon booting the kernel then loads only those drivers that are necessary for the specific hardware it is running on. We feel this scheme is less suited to embedded systems, where the hardware is known in advance and the necessary drivers can easily be compiled into the kernel, obviating the need for the module loading code altogether and thus reducing the kernel's code size. Furthermore, the module granularity is determined by the kernel's configuration system, and this is not fine-grained enough for our purposes. We wish to remove individual, infrequently executed, code paths from a driver or kernel subsystem, and not a driver or subsystem as a whole.

2.2 Our Solution

With our technique, which is only applicable to platforms that support virtual memory, the infrequently executed (henceforth called cold) code in the kernel is removed from physical memory, even though it is still present in the kernel's virtual memory image, and stored on a fast secondary storage medium. This storage is preferably Flash memory as this is already available in most embedded systems for storing the device's firmware. However, there are other possibilities, like storing the cold code in a compressed form in main memory (thus improving on [8]). When the kernel tries to execute the cold code, a page fault occurs. The (modified) page fault handler then locates the needed page in the secondary memory, loads it in one of a set of pre-allocated physical page frames and adjusts the page tables to map it to the correct virtual address, after which execution can continue. This basically means that the page fault trap is used as the trigger to load code.

The only modification necessary to the kernel source code is the extension of the page fault handler to insert the code loading mechanism. As such, we feel the transparency requirement has been fulfilled.

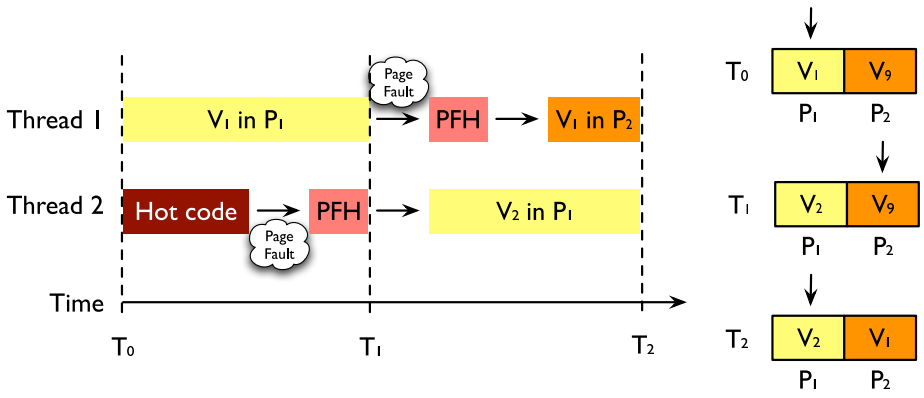

Fig. 1. An example of the concurrency issues involved in evicting code from memory. The left side of the figure shows a time line of the execution of two different threads, the right side shows the contents of the code cache at times T_0, T_1 and T_2. The downward-pointing arrow indicates the next page to be replaced according to the replacement policy.

In order to fulfill the "hard upper bound on memory usage" requirement, we use a fixed-size code cache of physical page frames to map the cold code. The cache is managed through some replacement policy (for instance, round robin) that does not explicitly check whether code on a page selected for eviction is being executed in another thread. Nevertheless, the reliability of the system is not compromised. This is illustrated in Figure 1. Assume there are two threads in the kernel, the cache can hold two pages, and we use a round-robin replacement policy. Thread 1 is executing cold code from virtual page V_1 in the physical cache frame P_1, while thread 2 is executing hot, non-swappable code. The second cache frame contains a previously loaded cold page V_9 from which no code is being executed any more. At some point, the second thread has to execute some cold code from virtual page V_2, which is currently not in the cache, causing a page fault. The page fault handler runs in the execution context of thread 2, locates the necessary code in secondary memory and because of the round-robin replacement policy decides to put V_2 in P_1. Once V_1 is unmapped from memory, a page fault occurs in thread 1 when the next instruction in this thread is loaded. The page fault handler runs in the context of thread 1, find V_1 in secondary storage and map it in P_2 because of the round-robin policy, after which execution in thread 1 can continue as before. While this scenario means that thread 1 has been temporarily interrupted, the integrity of the execution

has not been compromised. In the worst case, this scenario could cause a cascade of code cache refills for all kernel threads, but it is easily shown that the system will not deadlock as long as there are at least as many code cache frames as there are kernel threads.

There is only one manual step involved in selecting which code is loaded on demand: the profiling step. The user has to run an instrumented kernel on the target system in order to collect a basic block profile. This profile is then used to identify the cold code in the kernel, and from then on the whole process runs without user intervention. This satisfies the automation design criterium.

In order to fulfill the performance requirement, there are several issues we have to take into account. First, loading the page from secondary memory should be sufficiently fast. We believe this requirement to be fulfilled with the use of Flash memory as a secondary storage medium. For currently available Flash memory parts (Intel Embedded Strataflash P33), a 4 KiB page can be read in approximately 40 microseconds. As only code, which is read-only, is swapped in, there is no need to write back pages to Flash memory when they are evicted from memory, avoiding costly Flash write operations that would slow down the process. Secondly, only cold code should be swapped out in this way, to reduce the amount of needed code cache refills. Thirdly, an intelligent code placement algorithm should be used to avoid that related cold code fragments span page boundaries, because this would cause more code cache refills than necessary. Our code placement algorithms are detailed in Section 3.2.

Our approach is essentially a variation on the well-known virtual memory swapping technique [20]. Generic swapping can store any virtual memory page from any process on a secondary storage medium (typically a hard disk), thus freeing up physical memory for other virtual memory pages. Because of the high latency involved in reading a page from disk, the OS usually puts the process causing a page fault to sleep and schedules another process to run instead. This makes this technique less suitable for use in the OS kernel itself, and indeed Linux does not implement swapping for kernel memory. While this has been repeatedly proposed in the past, the kernel developers reject the idea because of the amount of timing-critical code in the kernel that cannot sleep. Separating this timing-critical code and data from other code and data would be too involved and error-prone to be practical [5].

We believe that our approach is not susceptible to these objections. As Flash memory is an order of magnitude faster than hard disks, there is no need to put the faulting execution thread to sleep, thus avoiding the problems usually associated with swapping out kernel memory.

Note that it would also be possible to extend this technique to the kernel's read-only data (i.e., strings for error description). However, we have left this problem for future work. Extending the technique to incorporate writable data is not advisable, as Flash memory wears down after too many write cycles. Consequently, the repeated write-back operations that swapping out writable data would entail, would severely limit the device's lifetime. Furthermore, the write-back operations would significantly slow down the swapping process.

3 Swappable Code Selection and Placement

In this section, we discuss how the code to be loaded on demand is selected, and we present the code layout algorithm that maps the swappable code to individual virtual memory pages in such a way that the need for code loading operations is minimized.

3.1 Code Selection

As mentioned in Section 2.2, only infrequently executed code is considered for on-demand code loading. Based on basic block profile information gathered for the kernel (the instrumentation technique is discussed in Section 4.2), the kernel code is divided into three categories:

1. The *core* code: this is the code that always has to be present in memory for the system to work correctly. Basically this portion of the code consists of all code that can be executed before our code loading mechanism is initialized, the page fault handling mechanism and the code needed to read the secondary storage medium.
2. The *base* code: this is the frequently executed (hot) kernel code, which we want to keep permanently resident for performance reasons, even though there are no technical difficulties in swapping it out.
3. The *swappable* code: this is the remaining code, which is either infrequently (cold) or never (frozen) executed. This is the code that is removed from the kernel image and stored on the secondary storage medium for on-demand loading.

It is important to note that by design the Linux kernel code is split into two categories: *initialization* code and *non-initialization* code, henceforth called *init* code and *non-init* code respectively. Because the kernel's first task at boot time is to initialize the system and create an environment in which application programs can run, it contains a lot of code and data that is only used once at boot time. As soon as this initialization code and data are no longer needed, the kernel removes them from memory. As such it is not very useful to apply the on-demand code loading technique to the init code: by the time the user space processes start executing, and the device's full memory capacity is needed, it is already removed from memory. Consequently, we consider all init code to be part of the core code.

As mentioned before, all code that is executed before our code loading mechanism is initialized has to be considered core code. While most of this code is init code, it also includes a number of non-init utility procedures that are called from the init code. We can reduce the amount of non-init core code by duplicating all non-init procedures that are only called from init code prior to the initialization of our code loading mechanism. All calls from init code are moved to the duplicate procedures, which can then be considered init code as well. The original procedures are then no longer called prior to the initialization of our

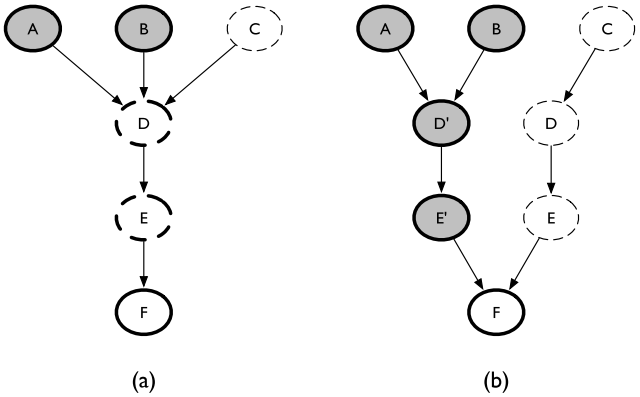

Fig. 2. A slice of the call graph (a) before and (b) after procedure duplication. Gray blocks represent init code, white blocks non-init code. Nodes with heavy borders are considered core code, those with a dashed border are infrequently executed, those with a solid border are frequently executed.

mechanism, and can be considered swappable. As the init code is released from memory during the boot process, the duplicated procedures incur no memory overhead during the system's steady state operation.

Figure 2 illustrates this process. In part (a) we see a slice of the kernel's call graph before duplication. Non-init procedure D is called by init procedures A and B before the code loading mechanism is initialized. The call from non-init procedure C can only occur after the mechanism is initialized. Because of the calls from A and B, D and its descendants in the call graph E and F must be considered core code. In part (b) the situation after duplication is shown. F is not duplicated as it is hot code, and is swapped out anyway. The duplicated procedures D' and E' are only reachable from init code, and can thus be considered init code themselves. The original procedures D and E can now only be called after the code loading mechanism is initialized and can hence be considered swappable instead of core code.

3.2 Code Placement

The idea of using code placement techniques as a means to minimize page faults has been studied before. An overview of the existing literature can be found in Section 6.3. All existing algorithms use some variation on run-time profile data as input, and of course they concentrate on achieving a good placement for the most-frequently executed code. As we only have to place the least-frequently executed code, for which there is much less profile information available, these algorithms are not guaranteed to achieve good results. This is especially true in the case where only frozen code is considered swappable, because for this code all execution counts are zero.

Therefore, we have implemented two different code placement algorithms. The first makes use of whatever profile information is available to achieve a good placement, whereas the second aims to minimize, for each entry point in the swappable code, the total number of pages needed to load all swappable code that is directly reachable from that entry point. The second algorithm makes no use of profile information and relies only on an analysis of the static structure of the code. Both algorithms assume that the swappable code can be placed independently from the hot and core code, i.e. there are no fall-through control flow paths connecting cold code to other code. How this is achieved in practice is explained in Section 4.3.

The profile-based algorithm In this algorithm, which is similar to the one proposed by Pettis and Hansen [23], the code is placed with a *chain* granularity. A chain is a set of basic blocks that have to be placed in a predetermined order because of control flow dependencies (e.g. fall-through paths or a function call and its corresponding return site). Control flow between chains is always explicit, in the form of function calls, returns or jumps. Consequently, the order of the chains is not important for the correct working of the code. Indirect control flow (i.e. indirect jumps and function calls) is not taken into account by the algorithm.

We use a graph representation of the problem as proposed by Ferrari [14]. The graph nodes represent chains. While hot chains will not be placed on the swappable pages, and their final layout will not be influenced by this algorithm, they are also represented in the graph. The size of a node representing a cold chain is equal to the size of the chain in bytes, whereas nodes representing hot chains have size 0. The (undirected) graph edges represent direct control flow between chains. The edge weights are computed by the following formula:

$$weight(e_{ij}) = \sum_{e \in (E_{i \to j} \cup E_{j \to i})} (1 + execcount(e))$$

where $E_{i \to j}$ is the set of direct control flow edges from chain i to chain j and $execcount(e)$ is the traversal count of control flow edge e according to edge profile information. In our current implementation, the edge profiles are estimated from the basic block profile information we have available. It is also possible to obtain exact edge profiles by inserting the appropriate instrumentation into the kernel, just like we did for obtaining the basic block profiles, but, as shown in the evaluation section, the estimated edge profiles are accurate enough to derive a good code placement. The traversal counts are incremented by one to ensure that the substantial body of frozen code in the kernel, whose edge traversal counts are zero, is not ignored during placement. If each node is placed on a separate virtual memory page, the graph's total edge weight is an estimate of the number of page faults that occur at run time.

The hot chains are represented in the graph to make sure that related cold code fragments are not placed independently. For example, suppose a procedure has a cold prologue and epilogue, but the actual procedure body is hot. As the

only way for the control to flow from the prologue to the epilogue is through hot code, the prologue and epilogue chains would not be connected in the graph if only the cold chains are represented. As a consequence, there is a big chance that the prologue and epilogue are placed on different code pages, which would result in two page faults for an execution of the procedure, as opposed to only one when they are placed on the same page.

The nodes in the graph are clustered in such a way that node sizes never exceed the virtual memory page size. This is done in three steps:

1. We try to minimize the total edge weight of the graph. This is done with a greedy heuristic by iteratively selecting the heaviest edge whose head and tail can still be merged without exceeding the page size. In case of a tie, we select the edge with the maximum *commonweight*, which is defined as:

$$commonweight(e_{ij}) = \sum_{k \in succ(i) \cap succ(j)} (weight(e_{ik}) + weight(e_{jk}))$$

   where $size(i) + size(j) + size(k) \leq$ PAGESIZE. In this way, we try to obtain a graph with less, but heavier edges instead of one with many light edges. If there still is a tie, we select the pair of nodes that exhibit the best locality, i.e., the pair of pages that contain code that was placed closest together in the original kernel. The intuition here is that code that was placed closely together is likely to be related. After this step, the total edge weight cannot be reduced any further.
2. We try to maximize the weight of individual edges by iteratively merging sibling nodes (nodes not connected to each other but connected to a common third node). In each iteration we select the nodes for whom the sum of the weights of the edges connecting them to their common parent is maximal. The idea behind this step is that, if more than one page is available in the code cache, the probability of page j already being in the cache upon a control transfer from page i is proportional to $weight(e_{ij})$.
3. For each connected subgraph, nodes are merged with a best fit algorithm. This step minimizes the total number of pages needed for each connected subgraph. We do not yet merge nodes from different subgraphs, because we do not want to pollute the pages for one connected subgraph with code from another subgraph. After all, the likelihood that node j is needed in memory before node i is removed from the code cache is higher if i and j belong to the same connected subgraph.

The per-entry point minimization algorithm This algorithm makes no use of profile information to guide the code placement. The swappable code is first partitioned into single-entry regions that do not span procedure boundaries. These single-entry regions (henceforth simply called regions) are the basic units of code placement. Regions that have incoming control flow edges from base or core code are called *entry points*. As a simplification, we assume that the entry points are independent of each other, i.e., that the fact that entry point i was

entered at time T has no influence on the probability of any specific entry point j being entered at time $T' > T$. Under this assumption, it makes sense to place the code in such a way that only a minimal amount of pages is reachable from each entry point. After all, in the absence of meaningful profile information we have to assume that all code paths through cold code are equally likely to be followed, so we cannot favor one code path over another for placement on a minimum number of pages.

Initially, each region is placed on its own page. Let P be the set of pages, and E the set of entry points ($E \subseteq P$). We define two functions:

$$\forall p \in P : \mathit{entries}(p) = \{e \in E \,|\, p \text{ is reachable from } e\}$$

and

$$\forall e \in E : \mathit{pcount}(e) = \sharp\{p \in P \,|\, e \in \mathit{entries}(p)\} \ .$$

$\mathit{entries}(p)$ returns the set of entry points from which code on a page p is reachable, *without passing through hot or core code*. $\mathit{pcount}(e)$ computes the number of pages that are reachable from entry point e.

The code placement algorithm tries to minimize the *pcount* for each entry point by iteratively executing the following steps:

1. Build the set M containing the entry points with maximal *pcount*.
2. Select pages p_i and p_j such that $\mathit{size}(p_i) + \mathit{size}(p_j) \leq$ PAGESIZE and p_i and p_j have a maximum number of entry points in common with M and each other, i.e. $\sharp(M \cap \mathit{entries}(p_i) \cap \mathit{entries}(p_j))$ is maximal. In case there are multiple eligible pairs, select the pair that has the most entry points in common. Stop if no pair can be found.
3. Merge pages p_i and p_j.

Reducing fragmentation Both described code placement algorithms terminate with a lot of small pages left that aren't merged because there are no direct control flow edges between the code on the pages. In order to reduce fragmentation, a post-pass merges these small pages. This happens in two steps:

1. Small pages that contain code that was placed close together in the original kernel are merged. If the code fragments originally were placed close together, they probably originate from the same source code file, which greatly increases the likelihood of the code fragments being related to one another.
2. The remaining pages are merged using a best fit approach.

4 Implementation

In this section we discuss the actual implementation of our technique. The binary rewriting operations are implemented with a modified version of the Diablo link-time binary rewriting framework that is suitable for rewriting Linux kernels [8]. The rewriting intervenes in the kernel build process just after all object files

are linked together to create the executable kernel image (the so-called *vmlinux* file). Note that this is not the last step in the build process, as this executable image is usually transformed into a self-extracting executable (the *bzImage* file) to save disk space and reduce the kernel's load time.

4.1 Linux/i386 Virtual Memory Management

Before describing our implementation, we review, focusing on the i386 architecture and Linux, address spaces, address translation, the organization of the virtual address space and page fault handling.

Address Spaces and Address Translation The i386 architecture supports 32-bit virtual address spaces and a 32-bit physical address space. Although extensions to i386 support bigger spaces, these are not relevant to the embedded context and are henceforth ignored. Virtual to physical address translation is performed by a paging mechanism. Pages are either 4 KiB or 4 MiB in size. The page table is organized as a two-level structure:

- The first-level page table (Page Directory) is recorded on a 4 KiB page, consisting of 1024 32-bit entries. Each entry contains a base physical address and some flag bits (present (valid), access rights, accessed, dirty, page size, ...). Each valid entry with the page size flag set represents a 4 MiB page; the remaining valid entries point to second-level page tables.
- Each second-level page table is also comprised of a 4 KiB page divided into 32-bit entries. These entries are similar to those of the Page Directory, except for the absence of the page size flag. Each valid entry of the second-level page table represents a 4 KiB page.

Each process has its own Page Directory. The `cr3` control register contains the physical address of the Page Directory of the running process. On each context switch, it is updated by the operating system.

Accessing either a Page Directory entry or a second-level page table entry whose present (valid) flag is not set triggers a page fault exception. When this happens, the virtual address that produced the fault is loaded into the `cr2` control register and a code that reflects the exception cause is pushed on the stack.

To speed up address translation, i386-family processors implement TLBs (Translation Lookaside Buffers) that keep some page translations in a cache memory. Storing a value in the `cr3` control register flushes the TLBs. The TLBs can also be flushed with a special instruction.

Organization of the Virtual Address Space Linux divides the 32-bit virtual address space into the user space and the kernel space. While each active process has its own user space, the kernel space is shared among all the processes.

The user space takes up the first 3 GiB of the virtual address space, up to address `0xbfffffff`. The fourth gigabyte of the address space is designated the

kernel space. The organization of the kernel space is dependent on whether the physical memory size exceeds 896 MiB. As this is not the case in the memory-constrained embedded devices we target, we ignore this possibility. The kernel space is divided into three areas:

- *Physical Memory Mapping*: This area provides a one-to-one mapping of physical to virtual addresses. Each physical address `pa` is mapped to virtual address `0xc0000000 + pa`. Contiguous virtual pages in this area are also contiguous in physical memory. The page table entries related to this area are initialized at boot time.
- *vmalloc*: This area avoids external fragmentation when the kernel allocates a contiguous multiple-page virtual space; contiguous pages in this area of the virtual space are not necessarily contiguous in the physical space. The area starts at the virtual address `VMALLOC_START` (typically 8 MiB after the end of the physical memory mapping) and ends at `VMALLOC_END`. Each allocation in this area is separated by a 4 KiB safety gap from the previous allocation. That is, the virtual pages adjacent to each allocation in this area are tagged as invalid in the page table. The page table entries related to this area are dynamically initialized as the `vmalloc` routine is called.
- *Fix-mapped addresses*: This area allows mapping a virtual page to an arbitrary physical frame. This area is placed almost at the end of the virtual address space. Fixed mappings are retrieved with the `fix_to_virt` routine. At compile time, the compiler is able to substitute all valid references to this routine with the corresponding virtual addresses. It is useful for subsystems that need to know the virtual addresses at compile time. The page table entries related to this area are dynamically initialized as the `set_fixmap` routine is called.

Figure 3 shows a diagram with the location of the three areas and the security gaps. We assume that `vmalloc` has been called twice, the first call has allocated 8 KiB and the second 4 KiB.

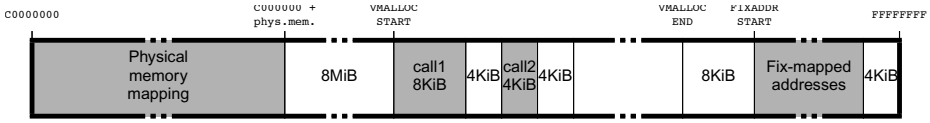

Fig. 3. Organization of the kernel address space.

Note that the Page Directory for each process maps both the user and kernel spaces. Moreover, the kernel maintains a Page Directory called the Master Page Table that is initialized, at boot time, with the physical memory mapping. During system operation, the kernel portion of a processes' Page Directory is initialized from the Master Page Table, and the Master Page Table is updated as the kernel's memory management routines are called.

Page Fault Handling The use of page fault exceptions on Linux depends on which portion of the virtual address space (user or kernel) the address belongs to.

In the user portion of the address space, page fault exceptions are mainly related to triggering code cache page refill. However, other scenarios can also produce these exceptions: copy-on-write handling, dynamic loading of the binary file, stack growth and detecting invalid memory accesses.

In the kernel portion of the address space, page fault exceptions are not related to code cache refills because the whole Linux kernel is permanently resident in physical memory. However, the kernel portion of the virtual address space can be modified dynamically through several kernel routines. Although these modifications are reflected in the Master Page Table, they are not propagated into the kernel portion of the Page Directory of all user space processes. Consequently, if the processor is running in privileged mode using the Page Directory of the running process, an exception may arise when accessing a kernel area outside the physical-memory mapping. Then, the page fault handler is responsible for synchronizing the contents of the Master Page Table with the process' Page Directory. Note that the second-level page tables related to the kernel space are shared by the kernel and by all processes, so those need not be synchronized.

4.2 Gathering Profile Information

As mentioned in Section 3, we need accurate basic block profile information to select the code to be loaded on demand. To collect this information, we generate an instrumented version of the kernel that is run on the system under typical workloads. The instrumentation added to the kernel is very straightforward: an extra zero-initialized data section is added to the kernel that contains a 32-bit counter for each basic block in the kernel. At the beginning of each basic block we insert an `inc $counter` instruction to increment the counter for that block. As the `inc` instruction affects the processor's condition flags, we need to make sure that the original flags are restored if their value is still needed afterwards. This can be determined using interprocedural register liveness analysis, which is already provided by the link-time rewriting framework we use. If the flags need to be preserved, a `pushf` instruction is added before and a `popf` instruction after the `inc`.

There are no special accommodations for reading out the counter values. The Linux kernel already offers the possibility to access the contents of the kernel's memory through the `/proc/kcore` interface, so the counter values are read directly from this interface.

In the next step, where the swappable code is separated from the always-resident (base and core) code, the basic block profile information is used to distinguish hot code from cold code based on a user-configurable threshold value T. For example, for $T = 0.95$, the most-executed basic blocks that together constitute (approximately) 95% of the kernel's execution time is considered hot, hence going into the base code partition. The hot code is identified with the following algorithm:

1. Compute the control flow graph's total weight. The total weight is defined as $W = \sum_{i=1}^{n} weight(block_i)$, where n is the number of basic blocks in the graph, $weight(block_i)$ is the execution count of the ith block multiplied by its number of instructions.
2. Sort the basic blocks on execution count in descending order.
3. Walk the sorted block list, summing the block weights until the accumulated weight is higher than or equal to $T * W$.
4. The control flow graph's *hotness threshold* H is then equal to the execution count of the last-visited block. All blocks whose execution count is higher than or equal to H are considered hot, all other blocks are cold.

Note that this algorithm is not exact, though the approximation it provides is sufficiently accurate to be useful.

4.3 Rewriting the Kernel

Our binary rewriter builds a control flow graph of the complete kernel, as described in [8]. On this graph, some preliminary optimizations are performed to reduce the kernel's memory footprint. First, all unreachable code and data are removed from the kernel. Next, using a technique described in [7], all non-init code that is only reachable from the init sections (and is thus unreachable after the init code is removed from memory) is identified and moved to the initialization code section. In this way, we minimize the amount of code that has to be considered for swapping.

Based on the profile information, the remaining non-init code is divided into core, base and swappable code. The swappable code is first partitioned into single-entry regions, which are then made individually relocatable. This means there are no control flow dependencies (such as fall-through paths or function call and return pairs) between the different regions or between swappable and always-resident code, so we can freely move the regions to new virtual addresses. To make the regions individually relocatable, we just break up all fall-through paths in and out of the regions by inserting direct jump instructions.

The swappable code is then partitioned into page-sized clusters according to the algorithms described in Section 3.2. Each cluster is then placed in a new code section, which is padded up to the 4KiB boundary. The rewriter's code layout phase then places these code sections in the virtual address region reserved for the code cache and adjusts all jump offsets and addresses in the kernel accordingly.

The Linux kernel code assumes that virtual addresses belonging to the Physical Memory Mapping area can be translated to physical addresses just by subtracting the constant `0xc0000000`, that is, Linux assumes that this virtual address range is always present in physical memory. Consequently, it makes no sense to place the code cache in this virtual address range as we cannot free the corresponding physical memory. Instead, we decided to place the code cache outside the Physical Memory Mapping area, at the end of the vmalloc area.

After the kernel image is emitted as an ELF executable, a simple GNU `objcopy` script extracts the swappable pages from the image and places them

in a second file. The remaining kernel image, which now no longer includes the swappable pages, is then used to generate the `bzImage` file. The swappable pages are stored in a separate partition of the device's Flash memory, and the generated `bzImage` is installed and booted just like a regular kernel.

4.4 The Modified Page Fault Handler

Our implementation is embedded into a driver statically linked in the Linux kernel. The driver's `init` procedure is called by the `init` kernel thread. This procedure saves the address of the original page fault handler and replaces it with a new handler, whose functionality is described later in this section. Also, the initialization procedure reserves a virtual address range at the end of the vmalloc area for the code cache. Moreover, it initializes the page table entries corresponding to the swappable code pages in the Master Page Table. Although the Linux kernel makes use, when possible, of 4 MiB pages, we should split the 4 MiB pages related to the swappable code because our implementation works on a 4 KiB page granularity. We create second-level page tables and initialize their entries as not present. Splitting 4 MiB pages can affect the TLB hit rate. Assuming that the kernel code size is smaller than 4 MiB, a TLB entry related to a 4 MiB page table entry maps the entire kernel code space. Note that these modifications to the Master Page Table are performed before creating any user process, so it is not necessary to propagate them to any processes' page tables.

When a page fault occurs, our new page fault handler checks if the virtual address responsible for the fault belongs to the swappable code address range. If that is not the case, control is handed over to the original page fault handler. Otherwise, our handler deals with the fault: a page is allocated in the code cache, the corresponding page is copied from secondary storage to physical memory, the corresponding second-level kernel page table is updated accordingly, and the faulting instruction is re-executed. Note that swapping cold pages in and out of memory does not affect the first-level page tables. It only affects the second-level kernel space page tables, which are shared among the kernel and all processes.

We have implemented three basic replacement algorithms: round robin, random and not recently used (NRU). The latter is implemented by periodically resetting the `accessed` flag of the page-table entries and flushing the TLB. When the cache is full, the page to be evicted is chosen randomly from those that have an unset `accessed` flag.

4.5 Portability

The proposed technique is easily portable to other operating systems. The only prerequisite for the OS kernel is that it supports virtual memory, and that it is possible to adapt the page fault handler so that it loads the cold code from the repository. We have studied the source code of FreeBSD (a Unix-like OS) and ReactOS (an open source Windows NT clone), and in both cases we were able to easily identify the page fault handler routine in which to insert our page loading code.

Of course, to gather the profile information and to split the cold code from the unswappable code, one also needs to have a binary rewriter that is capable of rewriting the OS kernel. While our binary rewriter is only capable of rewriting Linux kernels, it does not rely on any specific Linux concepts to enable reliable binary rewriting. Consequently, we believe porting the binary rewriter to other OS kernels is only an implementation challenge, not a conceptual one.

5 Evaluation

In this section, we first describe the environment used in our evaluations. Second, we evaluate the partitioning algorithms proposed in this work. Finally, we explore several dimensions of the design space of the proposed mechanism (code cache size and page replacement algorithm).

5.1 Evaluation Environment

The evaluation has been carried out using the 2.4.25 Linux kernel on an i386 system, but our mechanism can easily be applied to other operating systems on other platforms with only minor changes.

The benchmark suite used to stress the system is Mediabench II [1], which is suitable for testing embedded systems since it is composed of very specific multimedia applications. The full set of programs is: cjpeg, djpeg, h263dec, h263enc, h264dec, h264enc[3], jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc, mpeg4dec, and mpeg4enc. Each program has been executed 5 times in order to obtain more accurate results. The used input datasets are the ones bundled with the source code of the benchmark suite.

Basic block profiles have been collected by running the full benchmark suite on the target machine. In order to reduce external activities in the system, all the user-space daemons have been stopped. This way, most of the kernel code used by the benchmark applications is considered hot, and is hence placed in the base partition. There could be some noise in the profile since it is collected from the boot of the system to the end of the benchmark execution; therefore some code only executed during the boot could be considered hot although it might not be used by the benchmark applications.

The performance metrics considered in our evaluations are the kernel code memory footprint, the number of page faults caused by our mechanism, and the system-mode execution time observed during the sequential execution of all the applications in the benchmark.

The used hardware platform is an i386-compatible VIA C3 processor clocked at 1200Mhz, 256 MiB of RAM (limited to 64 MiB as we explain later), a VT823x chipset, and an IDE UDMA2 hard disk. As we did not have access to an embedded system with integrated Flash memory at the time of writing, we cannot give accurate measurements for the slowdown incurred by the code loading operations. However, we believe that we can still provide very accurate estimates of

[3] H264 encoder has been excluded due to its excessive execution time

the slowdown by inserting a realistic delay in the code loading mechanism that simulates the latency caused by reading a 4 KiB page from Flash memory.

In our current implementation, the swappable code resides in RAM, but in a physical range that is not used by the kernel (the kernel's physical memory usage was limited through the `mem` command line parameter at boot time). The swapped-out code is loaded into this physical range by a modified version of the GRUB boot loader. Loading code into the kernel-visible physical memory is then simply implemented by copying the appropriate page from this memory range into the physical code cache frames.

The latency that would be incurred by loading from a Flash device is simulated by a delay that is inserted in the copying code. In order to get a good estimate for this delay value, we have measured the time needed to read a 4 KiB block from Flash memory on a real embedded device. For this measurement, we have used an Intrynsic CerfCube 255, with a PXA255 XScale (ARM-based) processor clocked at 450 Mhz and 32 MiB of Intel StrataFlash J3 NOR-based Flash ROM. On this system, reading a contiguous 4096-byte block from Flash takes approximately 442 μs. Consequently, we have used a delay value of 442 μs for our measurements.

5.2 Results

We first investigate the impact of the hot code threshold value T (see Section 4.2) on the kernel code size. Next, we study the effect of the different code placement strategies described in Section 3 on the kernel performance. Finally, we study the impact of the code cache size and the page replacement policy on the performance, given a fixed code placement algorithm and a fixed value of the threshold T.

Name	Code size	Data size	System ex. time	User ex. time
Original	657	312	7.00	395.76
Minimized non-init code	581	286	7.27	395.60

Table 1. Original kernel characterization (sizes are in KiB and times in seconds)

Table 1 summarizes the characteristics of both the original kernel and one in which the amount of non-init code has been minimized by removing all unreachable code and data and moving non-init code to the init sections where possible, as described in Section 4.3. The table shows the sizes of the non-init code and data and the system mode and user mode execution time for one run of the benchmark set. The system-mode execution time for the kernel with minimized non-init code is 4% higher than for the original kernel. This difference can be ascribed to the different code layouts in both kernels, which has an effect on I-cache utilization, and thus results in small variations in execution time.

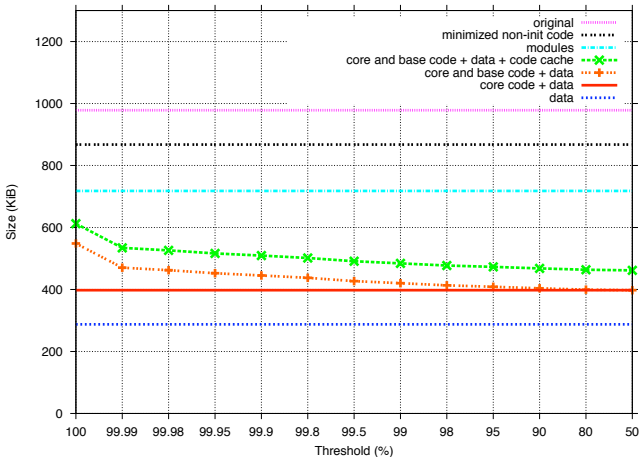

Fig. 4. Static non-init kernel footprint per threshold with and without a 16-page code cache

Influence of the hot code threshold on kernel code size The graph in Figure 4 shows the kernel's static memory footprint in function of the hot code threshold T, both with and without a 16-page code cache included. The sizes shown here do not include the initialization code and data, as these are removed from memory during the boot process and as such are not relevant for the kernel's memory usage during the time the system is actually in use. For reference, the footprint of the original kernel, one with minimized non-init code and one with all optional functionality compiled as modules are also shown. Note that the *modules* line only refers to the size of the image of a kernel compiled with module support but it does not include the footprint of any of the modules, so it marks the lower bound of its code footprint. Some of the functionality provided by the modules is always needed (e.g. root filesystem, disk device driver) and they are almost permanently resident in memory. Other modules are loaded and unloaded as their functionality is used (e.g. ELF loading). Therefore, during normal operation, the actual code footprint is higher since some modules are resident in memory.

While the static non-init footprint of the original kernel is 978 KiB, it is reduced to 867 KiB in the kernel with minimized non-init code, and to 718 KiB in a kernel compiled with module support. With the cold code removed from memory, the footprint ranges from 548 KiB ($T = 100\%$) to 398 KiB ($T = 50\%$). Taking into account a realistic code cache size of 16 pages, this becomes 612 KiB to 462 KiB, which amounts to a gain of 37.5% to 53.8% respectively when compared to the original kernel. Looking only at the non-init code sizes, the gains are 53% (code + code cache size 324 KiB) to 74.8% (code + code cache size 174 KiB) respectively.

As Figure 4 clearly shows, the interesting range for T values lies between 100% and 98%. Swapping out code at lower threshold values has almost no impact on the footprint reduction, whereas that it does have a significant impact on the kernel's performance, as we show later.

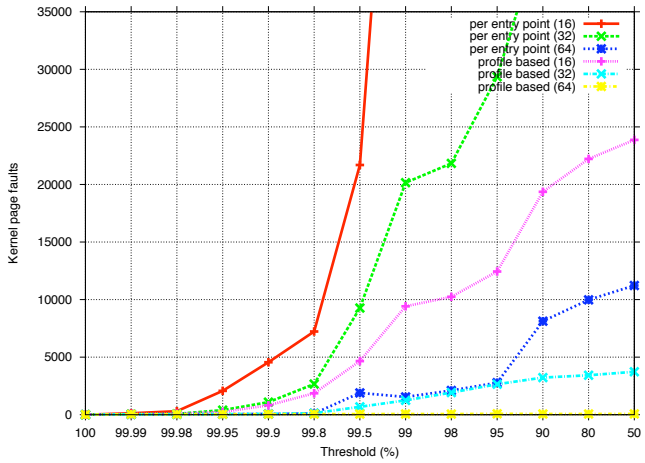

Fig. 5. Kernel code page faults for the different code placement strategies

Code partitioning evaluation In order to evaluate the effectiveness of the code placement strategies discussed in Section 3.2, we have generated kernels for the same threshold values as used in Figure 4, for each combination of the placement strategies (profile-based and per-entry point) and three different code cache sizes (16, 32, and 64 pages). For each of these kernels, we recorded the total number of code loading events (i.e., kernel space page faults, Figure 5) and the total system-mode execution time (Figure 6) during a run of the benchmark suite. Pages in the code cache are replaced according to an NRU policy. The user-mode execution times for the benchmark runs remained largely the same as for the original kernel, and are not shown here.

As can be expected, the profile-based code placement strategy is much more effective than the per-entry point strategy for lower values of T: the hotter the code that is swapped out, the more sense it makes to rely on profile information to guide the placement. For threshold values lower than 99.95 (using a 16-page or 32-page code cache), the number of page faults shown by the per-entry point algorithm grows exponentially. On the other hand, the profile-based strategy shows acceptable results even for a 16-page code cache. Using a 64-page code cache shows a reasonable number of page faults for the per-entry algorithm, although it remains much lower and almost constant for the profile-based algorithm.

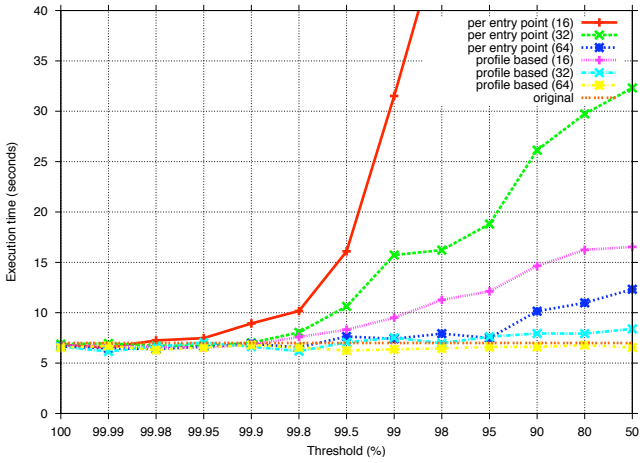

Fig. 6. System-mode execution time of the benchmark suite for the different code placement strategies

For reference, Figure 6 also includes the system-mode execution time of the benchmark suite on the original kernel (7 s). Interestingly, when the number of page faults is low enough (less than 1000) the kernel with swapping enabled outperforms the original kernel for all placement strategies. This can be attributed to a better I-cache utilization, as in the rewritten kernels the hot code is separated from the cold code, which is the basic concept behind code layout optimizations for improved cache utilization like the one proposed by Pettis and Hansen [23]. For lower values the per-entry point algorithm shows a large performance degradation caused by the huge number of page faults, while the profile-based algorithm obtains much better results, especially when using a 32-page (maximum slowdown of 1.3 s or 19%) or 64-page code cache (speedups from 0.7 to 0.2 s – 10% to 3%).

These results suggest that it is not very useful to use T values lower than 99.9%, as there is only little code size to be gained. With $T = 99.9\%$, a 16-page cold code cache suffices to get adequate performance: a code footprint reduction of 68% is coupled with a slight speedup of 2.2% in the system-mode execution time.

Design space exploration Finally, we explore the impact of the in-kernel eviction policy on the performance of the code-loading scheme. For these experiments, we focused on a kernel rewritten using the profile-based code placement strategy and a hot code threshold value $T = 99.9\%$.

Like in the previous graphs, code cache sizes are varied between 16, 32 and 64 pages, the following cache eviction policies are tried: round robin, random replacement and not-recently-used (NRU) replacement.

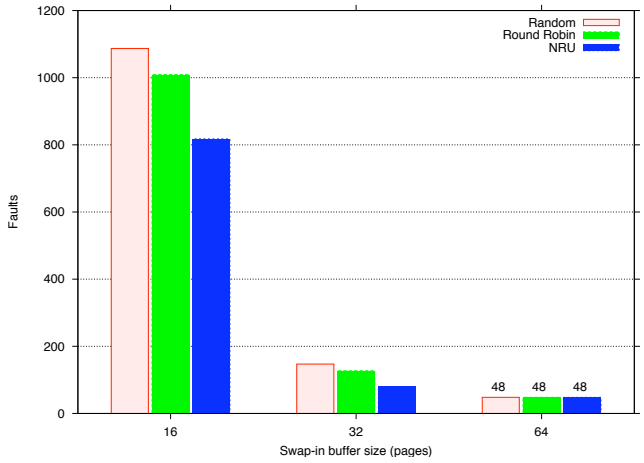

Fig. 7. Influence of the code cache size and the cache eviction policy on the amount of kernel code page faults

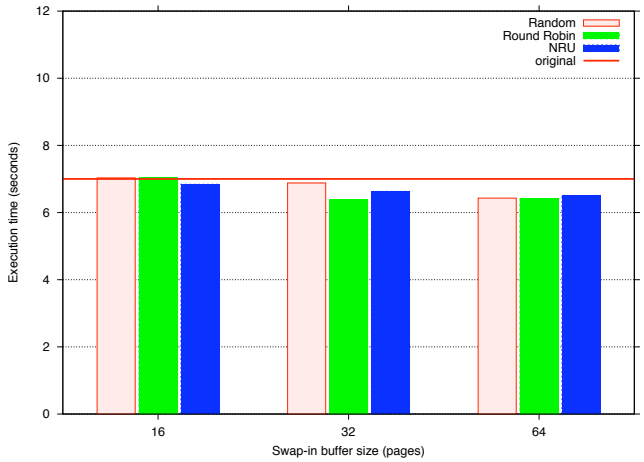

Fig. 8. Influence of the code cache size and the cache eviction policy on the system-mode execution time

First, the influence of the code cache size and the cache eviction policy on the number of page faults is shown in Figure 7. We observe that a 64-page code cache holds the entire working set of swappable code pages. Decreasing the code cache size significantly increases the amount of kernel page faults. However this number of faults is not enough to significantly penalize the system level execution time, as shown in Figure 8. There is a slight speedup when compared to the original kernel on all our tests using different code cache sizes. We presume this to be caused by

the aforementioned I-cache utilization effects. The difference in execution times for the different combinations of cold code cache size and replacement algorithm is proportional to the measured number of page transfers.

Focusing on the cache eviction policy, we observe that NRU always performs best when looking at the number of page faults. The same goes for system-mode execution times, with the exception of the 64 page case. There, as the whole working set of the kernel fits into the code cache, the number of page faults remains constant for the 3 eviction policies.

Impact on the performance of I/O-heavy applications For each kernel page fault, a 4 KiB page has to be read from Flash. One might wonder in which way this affects the performance of user space programs that make heavy use of the Flash memory for loading and storing data. Fortunately, the impact of the code loading on the total available Flash bandwidth is small. Assume that we have chosen a threshold of $T = 99.9\%$, with a 16-page cold code cache and the NRU replacement policy. For our test environment, this boils down to a total of 818 page faults over the execution of the benchmarks. Each page fault takes approximately 442 μs, so the total time spent reading cold pages from Flash is approximately 0.36 s. Given the total execution time of the benchmarks (user time + system time) of 403.2 s, this means that the code loading takes up only 0.09% of the total Flash bandwidth.

6 Related Work

In this section, we will discuss the related work with regards to OS kernel memory footprint reduction, on-demand code loading and minimization of page fault occurrences through code reordering.

6.1 Operating System Memory Footprint Reduction

The idea of specializing the Linux kernel for a specific application was first explored by Lee et al. [21]. Based on source code analysis, a system-wide call graph that spans the application, the libraries and the kernel is built. On this graph, a reachability analysis is performed, resulting in a compaction of a Linux 2.2 kernel of 17% in a simple case study.

Chanet et al. [7] use link-time binary compaction techniques to reduce the memory footprint of the Linux kernel. For systems that have a known, fixed hardware and software configuration, several specialization techniques that reduce the memory footprint even further are introduced. Run-time static memory footprint reductions (that is, not counting the kernel's dynamically allocated memory) of about 16% were achieved for Linux 2.4 kernels compiled for the ARM and i386 architectures.

He et al. [19] use similar binary rewriting techniques to reduce the code size of the Linux kernel. A novelty in their approach is the use of *approximate decompilation* to generate C source code for hand-written assembly code in the

kernel. This allows the use of a source code based pointer analysis (the FA-analysis [22]) for the identification of targets of indirect function calls. While the generated source code is not functionally equivalent to the original assembler code, it exhibits the same properties with regards to this FA-analysis. On a Linux 2.4 kernel without networking support, they report a code size reduction of 23.83%. For the same test system as used by Chanet et al. the results roughly correspond to those from [7], suggesting the two techniques are approximately equal in strength.

Later work by Chanet et al. [8] extends on the previous techniques by means of code compression techniques. Through code coverage analysis, frozen code is identified. This code is then stored in compressed format and decompressed at run time only if it is actually needed. To avoid concurrency issues, once code is decompressed, it is never evicted from memory. This makes this technique useless for the more general cold code compression, as the cold code will be executed at least once, and thus still take up memory for the rest of the system's run time. However, even after compaction and specialization of the kernel, there is still over 50% frozen code in the kernel, making this technique worthwhile. The combined compaction, specialization and compression techniques reduce the static memory footprint of a Linux 2.4 kernel with 23.3% for the i386 architecture and 28% for the ARM architecture. The smaller kernels suffered from a performance degradation of 2.86% (i386) and 1.97% (ARM). The same evaluation systems were used as in [7], so the results can be compared directly.

An alternative approach to customize an OS for use in embedded devices is proposed by Bhatia et al. [6]. The authors of this paper propose to remotely customize OS modules on demand. A customization server provides a highly optimized and specialized version of an OS function on demand of an application. The embedded device needs to send the customization context and the required function to the server and on receipt of the customized version, applications can start using it. The size of the customized code is reduced up to a factor of 20 for a TCP/IP stack implementation for ARM Linux, while the code runs 25% faster and throughput increases by up to 21%.

While our approach to minimize the kernel's memory footprint is top-down in that we start with a full-featured kernel and strip away as much unneeded functionality as possible, there is a number of projects that take a bottom-up approach. The Flux OSKit [15], Think [13] and TinyOS [16] are operating system construction frameworks that offer a library of system components to the developer, allowing him to assemble an operating system kernel containing only the needed functionality for the system.

6.2 On-demand Code Loading

The technique proposed in this paper is of course very much influenced by the virtual memory techniques used in most modern processors and operating systems [20]. The most important differences are that we selectively swap only parts of the kernel code in order to reduce the number of necessary code cache refills and that most VM systems use the hard disk, which is very slow compared to

main memory, as the off-line storage medium. As such, swapping incurs much bigger latencies, that make the method less suitable for application in timing-critical programs like an OS kernel.

Citron et al. [9] propose to remove frozen code and data from the memory image of a program. Control transfers to frozen code and memory accesses to frozen data are replaced by illegal instructions. The interrupt that occurs on execution of these illegal instructions is then intercepted and used as a trigger to load the needed code or data. A size reduction of 78% for the MediaBench suite of programs is reported. As only frozen code and data are loaded on demand, there are very little load events necessary, which makes the performance impact negligible. The approach is conceptually similar to ours, with the illegal opcode exception replacing the page fault exception as the trigger to load new code. Because this approach is not bound to the VM system, it is possible to load code and data at smaller granularity than the 4 KiB blocks that we use. However, while the paper mentions that once-loaded code may be evicted when memory is low, the authors do not discuss the concurrency issues this entails in multi-threaded programs. As such, it is not clear how well their technique holds up for this kind of programs.

Debray and Evans [11] use software-controlled code compression to reduce the code size of programs for the Alpha architecture. With profile data infrequently executed code fragments are detected. These fragments are stored in memory in a compressed form, and replaced by stubs. Upon execution of a stub, the necessary code is decompressed in a fixed-size buffer. When applied to programs that were already optimized for code size using link-time compaction techniques [12], additional code size reductions of 13.7% to 18.8% were achieved. The performance impact ranges from a slight speedup to a 28% slowdown. Again, no attention is given to concurrency issues that may arise with code eviction in multithreaded programs.

6.3 Code Reordering for Page Fault Minimization

Hatfield and Gerald [18] describe a technique that aims to minimize the number of page faults for both code and data references. The code and data are divided into a set of relocatable blocks (e.g. an array or a procedure). Using profile data, a *nearness matrix* is constructed, with one row and column for each relocatable block. Entry c_{ij} of this matrix represents the count of references from block i to block j. Virtual memory pages correspond to square regions along the diagonal of the matrix. By reordering the rows and columns of the matrix, the largest entries are brought closest to the diagonal, which corresponds to placing the blocks that reference each other most on the same page.

Ferrari [14] formulates the problem as a graph clustering problem. Nodes in the graph represent relocatable blocks. The weight of a node equals the size of the block it represents. Edges in the graph represent interblock references, and can be weighted according to various cost functions. An optimal ordering is then sought by clustering graph nodes in such a way that no node becomes larger than the page size and the total weight of the remaining edges is minimal. If the edges

are weighted according to profile information, this method is equivalent to that of Hatfield and Gerald. The author proposes a better-performing edge weighting, however, that is based on a trace of the block references during execution instead of mere profile data.

Pettis and Hansen [23] propose to reorder the procedures in a program in such a way that those procedures that call each other most frequently are placed closest together. Their main aim is to reduce the number of conflict misses in the instruction cache, but they note that this placement algorithm also reduces the number of page faults during program execution. Once again, the program is represented as a graph, with the nodes representing the procedures, and edges representing procedure calls. The edges are weighted according to profile information. In each step of the algorithm, the edge with the highest weight is selected and its head and tail nodes are merged. This method does not prevent procedures from spanning page boundaries.

Gloy and Smith [17] also reorder procedures to improve a program's instruction memory hierarchy behaviour. Their technique is similar to Pettis and Hansen's, but instead of profile information they use *temporal ordering information*, which not only summarizes the number of calls from one procedure to another, but also in which way these calls are interleaved. While their approach is in the first place directed towards optimization of the cache utilization, the authors also discuss an extension of the technique to minimize the amount of page faults.

7 Conclusions & Future Work

In this paper we introduced a novel on-demand code loading technique aimed at reducing the memory footprint of operating systems kernels for embedded systems with support for virtual memory. The page fault mechanism provided by the processor is used to trigger code loading and to avoid concurrency issues related to eviction of already-loaded code from memory. By using profile information, we can limit the code loading scheme to infrequently executed code and thus limit the performance impact. For a case study involving the Linux 2.4.25 kernel on an i386 platform, we were able to reduce the static kernel memory footprint (code + data) with up to 54.5%, with a slight speedup of the system-mode operations of 2.2% for our best-performing code placement strategy. When taking user-mode execution time into account, the speedup drops to 0.04%, which is negligible. Consequently, the proposed technique is a viable means of reducing the memory footprint of an OS kernel for use in an embedded system.

We have investigated two different code placement strategies: one based on profile information and one that just takes the static structure of the code into account, and have shown that the profile-based strategy is always best, even when placing only cold code, for which there is little profile information available.

In future work, we will focus on improving the code placement strategies to reduce the amount of page faults that still occur, and extend the technique to on-demand loading of read-only kernel data sections.

Acknowledgments

The authors wish to acknowledge the HiPEAC European Network of Excellence for the support it has given to our research. The work of Dominique Chanet was funded in part by the Flanders Fund for Scientific Research (FWO-Vlaanderen). We would also like to thank Bruno De Bus for his excellent insights and suggestions.

References

1. Mediabench II benchmark. `http://euler.slu.edu/~fritts/mediabench/`.
2. The Intel XScale microarchitecture technical summary. `http://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf`.
3. MIPS32 4Kc processor core data sheet. `http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorCores/4KFamily/MD00039-2B-4KC-DTS-01.07.pdf/getDownload`.
4. The Texas Instrument OMAP platform. `http://www.ti.com/omap`.
5. Discussion on kernel paging on the linux kernel mailing list, April 2001. `http://lkml.org/lkml/2001/4/17/115`.
6. Sapan Bhatia, Charles Consel, and Calton Pu. Remote customization of systems code for embedded devices. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 7–15, New York, NY, USA, 2004. ACM Press.
7. D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the Linux kernel. In *Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 95–104, 2005.
8. Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. Automated reduction of the memory footprint of the linux kernel. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(2), 2007. To appear.
9. Daniel Citron, Gadi Haber, and Roy Levin. Reducing program image size by extracting frozen code and data. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 297–305, New York, NY, USA, 2004. ACM Press.
10. B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems*, 27(5):882–945, 9 2005.
11. Saumya Debray and William Evans. Profile-guided code compression. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, New York, NY, USA, 2002. ACM Press.
12. S.K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 3 2002.
13. Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think: A software framework for component-based operating system kernels. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association.

14. Domenico Ferrari. Improving locality by critical working sets. *Commun. ACM*, 17(11):614–620, 1974.
15. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, New York, NY, USA, 1997. ACM Press.
16. David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.
17. Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.
18. D. J. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.
19. HaiFeng He, John Trimble, Somu Perianayagam, Saumya Debray, and Gregory Andrews. Code compaction of an operating system kernel. In *Proceedings of Code Generation and Optimization (CGO)*, March 2007. To appear.
20. John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*, chapter 5. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
21. C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee. An application-oriented Linux kernel customization for embedded systems. *Journal of Information Science and Engineering*, 20(6):1093–1107, 2004.
22. Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engg.*, 11(1):7–26, 2004.
23. Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.