The Revenge of the Overlay: Automatic Compaction of OS Kernel Code via On-Demand Code Loading[*]

Haifeng He, Saumya Debray, Gregory Andrews
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
{hehf, debray, greg}@cs.arizona.edu

ABSTRACT

There is increasing interest in using general-purpose operating systems, such as Linux, on embedded platforms. It is especially important in embedded systems to use memory efficiently because embedded processors often have limited physical memory. This paper describes an automatic technique for reducing the memory footprint of general-purpose operating systems on embedded platforms by keeping infrequently executed code on secondary storage and loading such code only if it is needed at run time. Our technique is based on an old idea—memory overlays—and it does not require hardware or operating system support for virtual memory. A prototype of the technique has been implemented for the Linux kernel. We evaluate our approach with two benchmark suites: MiBench and MediaBench, and a Web server application. The experimental results show that our approach reduces memory requirements for the Linux kernel code by about 53% with little degradation in performance.

Categories and Subject Descriptors: D.3.4 [Programming Language]: Processors–code generation, compilers, optimization; D.4.2 [Operating Systems]: Storage Management–secondary storage, storage hierarchies

General Terms: Algorithms, Design, Experimentation, Performance.

Keywords: Code compaction, Code clustering, Binary rewriting, Embedded systems.

1. INTRODUCTION

Technological trends in recent years have led to the growing use of general-purpose operating systems, such as Linux, in embedded contexts. While this is in many ways a simpler and more economical approach to managing operating systems for embedded devices, it has the disadvantage that such operating systems—precisely because they are general-purpose—contain features and code that are not needed in every application context, and which incur unnecessary overheads, e.g., in execution speed or memory footprint. Such overheads are especially undesirable in embedded processors and applications because they usually have resource constraints, such as a limited amount of memory. This makes it important to try and reduce the memory footprint of embedded code—including the operating system—as much as possible.

This paper focuses on automatic techniques for reducing the memory footprint of operating system kernels in embedded systems. Prior work in this area has generally focused on in-memory techniques: eliminating unnecessary or duplicate code [3, 8] and compressing infrequently executed code for on-demand decompression [2] (Section 5 gives a more complete discussion of related work). These approaches are able to accomplish significant code size reductions: unnecessary code elimination achieves size reductions of 18%–24% [3, 8], while augmenting this with compression of unexecuted code gives an additional 7%–12% reduction in overall memory footprint [2]. However, these approaches have the limitation that all of the entire kernel code, whether or not it is compressed, is kept in memory, which can unnecessarily limit the amount of code size reduction achievable.

Keeping all of the kernel code in memory might be reasonable if most or all of the code was actually executed. It turns out, however, that (at least for embedded applications) most of the kernel code is executed infrequently, if at all. When running the MiBench embedded application suite [7] on a minimally configured Linux kernel, for example, we found that out of a total of 213,862 instructions (occupying a total of 633.7 KB of memory), only 71,298 instructions (occupying 202.8 KB of memory)—i.e., about 32%—were actually executed. However, although about 68% of the kernel code is not executed, existing code optimization techniques are able to prove only about 20% of the code to be unreachable; the remaining code cannot be discarded because in theory it could be executed depending on other inputs to the applications running. However, keeping this code in memory uses up a scarce resource.

While embedded systems typically have a limited amount of main memory, they often have a considerably greater amount of secondary storage available, e.g., in the form of flash memory. This paper describes an automated approach that takes advantage of this feature to reduce the main memory requirements of the OS kernel. The essential idea behind our approach is to keep rarely used code on secondary storage, and load it into main memory if and when it is needed. We apply clustering to the whole-program control flow graph of the kernel to group "related" code fragments together; this has the effect of dividing up the OS kernel code into partitions and allows us to reuse the main memory buffer, into which code is loaded on demand, for different partitions at different times. If

---

[*] This work was supported in part by NSF Grants CNS-0410918 and CNS-0615347.

EMSOFT'07 September 30–October 3, 2007, Salzburg, Austria.

necessary, a very minor modification of the kernel code suffices to deal with multi-threading issues. The result is reminiscent of the old idea of code overlays, except that in our case the entire process of clustering the code into partitions, and transforming it to manage the code overlays, is automatic. Our approach does not require hardware or operating system support for virtual memory. Therefore, it is also applicable to low-end embedded system without virtual memory or MMU hardware support. Experiments with the Linux kernel indicate that we are able to reduce the main memory code size requirements of the OS kernel by around 53% with very little degradation in performance.

2. BACKGROUND

OS kernel code is quite different from ordinary application code. A significant problem is the presence of considerable amounts of hand-written assembly code that often does not follow the familiar conventions of compiler-generated code, e.g., with regard to function prologues, epilogues, and argument passing. This makes it difficult to use standard compiler-based techniques for whole-system analysis and optimization of kernel code. To deal with this problem, we decided to use binary rewriting for kernel compaction. However, while processing a kernel at the binary level provides a uniform way to handle code heterogeneity arising from the combination of source code, assembly code and legacy code such as device drivers, it introduces its own set of problems. Since the topic of binary rewriting of operating system kernels is somewhat orthogonal to the topic of this paper, we only briefly summarize the issues that arise; this topic is discussed in more detail elsewhere [13].

The main differences between operating system kernels and ordinary applications fall under the following categories:

Data in text section. The Linux kernel contains a significant amount of data embedded within the text section. For example, in Linux version 2.4, the page tables are placed within the text section. Such embedding of data within the instruction stream complicates the disassembly process.

Hand-written assembly code. Unlike application programs, which are typically written in high-level languages such as C or C++, operating system kernels typically contain a significant amount of hand-written assembly code. This complicates the problem of program analysis.

Use of indirection. Operating system kernels often use indirect control transfers through function pointers to enhance extensibility and maintainability. This makes precise program analysis difficult.

Implicit Entry Points. In an ordinary application program, the entry point is well-defined. This simplifies the task of disassembly and program analysis. By contrast, operating system kernels have multiple implicit entry points in the form of system calls and interrupt handlers.

Implicit Control Flow. Operating systems contain implicit control flow paths, e.g., for exception handling, that are not readily evident from examining the code. This complicates the task of program analysis and transformation.

For the reasons given above, binary rewriting techniques applicable to application code do not always carry over directly to kernel code. They also mean that code optimization techniques developed for application code (e.g., some of the work on dynamic code loading from secondary storage discussed in Section 5) may not be directly applicable to OS kernel code.

3. CODE COMPACTION

As mentioned earlier, the main idea behind our approach is to keep the commonly executed OS kernel code (the "hot" code) in main memory while moving the rarely executed code (the "cold" code) to secondary storage, and with appropriate adjustments to load cold code into memory when (if) it is needed. The code transformation needed to accomplish this basic functionality is conceptually straightforward: on each control flow edge that goes from the hot code to the cold code, insert some code to load the cold code from secondary storage into memory; and adjust addresses to reflect the fact that the cold code may execute at a different location in memory than its original address when it is loaded. Note that while code for on-demand loading is inserted along control flow edges from the hot code in memory to the cold code in secondary storage, no corresponding code is needed on edges from the cold code to the memory-resident code. This is because the memory resident code always stays in memory, while the code loaded from secondary storage is not modified and so does not need to be written back to secondary storage.

This straightforward idea is much too simplistic, however. We have to reserve enough space in memory to accommodate any code that may have to be loaded; we call this region of memory the *code buffer*. If all we do is that described above, then the code buffer has to be large enough to accommodate all of the cold code, resulting in no net space savings (it would actually make things a little worse because of the additional code needed for on-demand loading of code from secondary storage).

It is evident from this discussion that we have to do three things in order to make our approach profitable. First, we have to determine how much code is to be memory-resident and how much is to be kept in secondary storage: increasing the amount of code moved to secondary storage leads to a reduction in the memory footprint of the kernel, but can potentially lead to runtime overhead increased because of the need to load code into memory during execution. Second, we have to load the cold code in smaller chunks so as to reduce the time and energy cost of loading code. Finally, the memory region where these code chunks are loaded has to be reused, i.e., the code buffer needs to be able to hold different chunks of code at different times. If the code loaded from secondary storage is partitioned into k chunks, of sizes $n_1, \ldots, n_k$, then this code buffer reuse makes it possible to use a buffer of size $max_{i=1}^{k} n_i$ instead of $\sum_{i=1}^{k} n_i$ as in the case with no reuse. However, these requirements raise some technical issues, which we discuss in the remainder of this section. We deal with first issue via a user-specifiable threshold that controls the amount of code that is memory-resident; this issue is discussed in Section 3.1. The second requirement implies that we have to be able to divide the cold code into different chunks in some reasonable way, so as to minimize the total number of loads from secondary storage; we discuss this in Section 3.2. The third requirement introduces some complications into the handling of control transfer between two different dynamically loaded code chunks; this issue is discussed in Section 3.3.

3.1 Upper-Bound of Memory Requirement for Kernel Code

The amount of memory required for kernel code is determined by two factors: the size of the code that is always kept in memory and the size of code buffer, i.e., the memory region that is used to keep the code that is loaded dynamically. The sum of these two values gives an upper bound on the memory usage for the kernel code.

In our approach, the code that is always kept in memory(memory-resident) consists of two categories:

```
Input:  A basic block $bbl$
Output: The estimated final size of $bbl$
Method:

    $N$ = the total memory size of original instructions
        in $bbl$.
    if $bbl$ contains a control flow edge to a different cluster or
        an indirect edge (means control target is unknown) do
        return $N$ + size increased due to code transformation.
            (see Section 3.3).
    else
        return $N$.
    fi
```

Figure 1: The $BlockSize$ function

- The *core code*, which is the code that has to be in memory to make the kernel work correctly. This includes the scheduler, the memory management, the trap and interrupt handling code in the kernel, and—in the final overlay-based version of our system—the code that manages overlay. Our current implementation identifies such code by having the user designate a specific set of kernel functions as core code.
- The "hot" code, i.e., frequently executed code that is kept in memory in addition to the core code for performance reasons. Our approach uses a user-specified parameter γ that determines how much code is kept in the memory. In our current implementation, γ is specified as the percentage increase allowable in the size of the core code relative to its size in the input binary. Thus, if core code occupies 100 KB in the input kernel binary and $\gamma = 10\% = 0.1$, then the memory-resident code in the transformed kernel is allowed to be up to 10% larger than the core code in the input binary, i.e., up to 110KB.

We assume that the size of code buffer is given as an input parameter (the experimental results reported in this paper were based on the size of code buffer = 2 KB). The size of code buffer, denoted as $BufSz$, limits how large each code chunk(cluster) can be in order to be accommodated in the code buffer. With parameter γ and $BufSz$, the upper-bound of memory usage for kernel code equals to

$$size(core\ code) \times (1 + \gamma) + BufSz.$$

3.2 Code Clustering

In order to minimize the number of reads for code from secondary storage, we use a greedy node-coalescing algorithm for clustering. We begin with an edge-weighted whole-program control flow graph for the kernel. The edge weights, representing execution frequency counts, are obtained via edge profiling. Since there is a significant amount of indirect control transfers through function pointers in the kernel, we also perform target profiling for all indirect control transfer instructions to collect the weights for indirect edges and add them into the whole-program control flow graph.

From the edge-weighted whole-program control flow graph, we construct an edge-weighted graph, the *cluster graph*, whose nodes represent the code clusters and whose edges represent control transfers between clusters.

The details of our clustering algorithm are shown in Figure 2. The algorithm operates on chains of basic blocks that must be contiguous in memory, e.g., due to fall-through edges or the return from a function call. There are four major steps in the algorithm:

1. *Create a cluster graph*

   Initially, each chain is assigned to a separate cluster; there is an edge e between two clusters a and b if there are any control flow edge between a and b, and the weight of the edge e is computed as the total weight of all the control flow edges between the blocks in a and b.

2. *Compute the code size and cluster size for each node*

   The cluster size of each node determines how much code a cluster can hold. For each node other than the node C for the core code, the cluster size is given by $BufSz$, the size of the code buffer. The cluster size of C is the final size of code that is always in memory. The upper-bound is determined by the size of *core code* and the code-size bound specified by the parameter γ.

   The code size for a cluster is computed as the total size of all of the basic blocks in that cluster. The function $BlockSize$, shown in Figure 1, is used to compute the memory size of a basic block. This is given by the total size of the instructions in the basic block together with the size of any additional code that is inserted to support overlays. The reason for the latter code is that if there is an inter-cluster control transfer in a basic block, the control transfer instructions in the basic block have to be modified to deal with overlays (see Section 3.3).

3. *Node coalescing*

   The algorithm then processes the cluster edges in descending order of weight, iteratively coalescing the end-points of edges whenever possible until no further coalescing can be carried out. Two nodes a and b can be coalesced if doing so will not cause the size of the resulting (coalesced) node to violate the following conditions:

   - If neither a nor b is the core node C, then the size of resulting node must not exceed the cluster size of either node, which is equal to $BufSz$.
   - If either of a, b is C, the size of resulting node should not exceed $\max(C.cluster\_size, BufSz)$. In this case, the code of the other node becomes "hot" code and memory-resident as well.

4. *Defragmentation*

   At the end of this step, there are usually some small clusters left over; a defragmenting step is carried out at the end to merge such clusters with larger ones where possible.

In the algorithm, the bound parameter γ controls the final size of memory-resident code. If $\gamma = 0$, only the core code will be kept in memory; all other code, including even hot code, will be kept in secondary storage, likely resulting in a large number of reads into the code buffer and a concomitant high runtime overhead. Larger values of γ mean that some additional code can be kept memory-resident; since we process cluster edges in descending order of weight, this will cause some of the frequently executed code (which must be small enough to fit into the additional memory space that is now available) to be coalesced with the cluster corresponding to the core code. This will result in reduced runtime overheads because less code will have to be loaded at runtime. Our experimental results, reported in Section 4, confirm this.

Input:

1. An edge-weighted control flow graph for a program, together with a function *BlockSize* that gives an estimate of total memory size of each basic block
2. A set of functions $\mathcal{F}$ that must reside in memory. The code for these functions comprises the *core code*.
3. A bound γ on the final size of the memory-resident code.
4. An integer $\mathit{BufSz} > 0$ giving the size of code buffer.

Output: A cluster graph for the program.

Method:

1. Create a cluster graph $G = (V, E)$ as follows:
   - V contains a single node C corresponding to the *core code*, as well as a node for each basic block chain B such that $B \neq C$.
   - There is an edge (a, b) between nodes a and b in the cluster graph if there is a control transfer edge between some basic block in a and some basic block in b. The weight $w(e)$ of an edge $e = (a, b)$ is given by the total edge weight of all control flow edges between blocks in a and those in b.
2. Compute the code size and cluster size for each node:
   - For each node a in the cluster graph, let $a.\mathit{code\_size} = \sum\{\mathit{BlockSize}(\mathit{bbl}) \mid \mathit{bbl} \in a\}$.
   - Let $C.\mathit{cluster\_size} = C.\mathit{code\_size} \times (1 + \gamma)$.
   - For each node $a \neq C$, let $a.\mathit{cluster\_size} = \mathit{BufSz}$.
3. [*Node coalescing.*] Process the edges of the cluster graph G in descending order of weight, iteratively coalescing nodes:

```
while $\exists (a, b) \in E$ s.t. $(a.code\_size + b.code\_size) \leq \max(a.cluster\_size, b.cluster\_size)$ do
    Coalesce the endpoints $a$, $b$ and merge $b$ with $a$, setting:
        $a.code\_size = a.code\_size + b.code\_size$;
        $a.cluster\_size = \max(a.cluster\_size, b.cluster\_size)$.
    Update edge weights for all clusters adjacent to $a$, $b$ appropriately.
od
```

4. [*Defragmentation.*] Coalesce small clusters into larger ones where possible:

```
while $\exists a, b \in V$ s.t. $(a.code\_size + b.code\_size) \leq \max(a.cluster\_size, b.cluster\_size)$ do
    Merge $b$ with $a$, setting:
        $a.code\_size = a.code\_size + b.code\_size$;
        $a.cluster\_size = \max(a.cluster\_size, b.cluster\_size)$.
od
```

5. Return the resulting cluster graph.

Figure 2: The Clustering Algorithm

3.3 Code Transformation

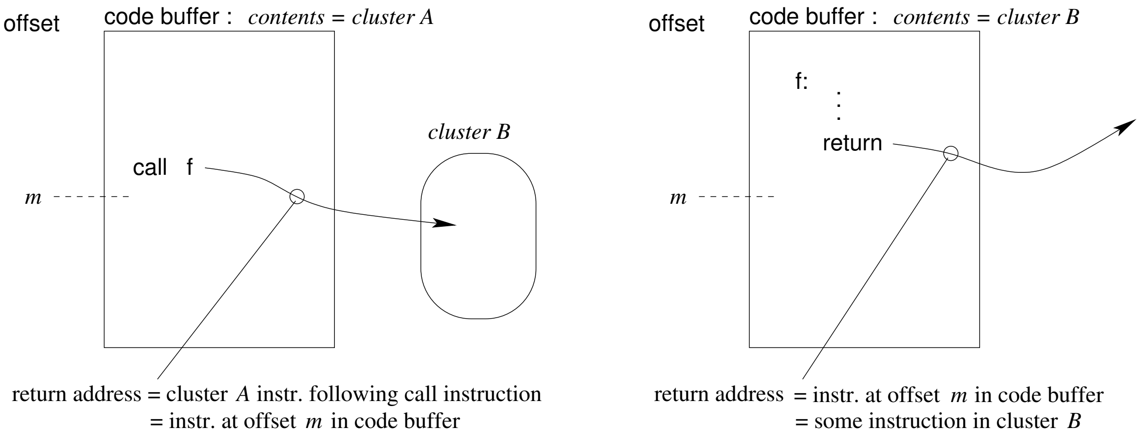
Once clustering has been done, the next step is to transform the kernel code to support overlays. We add a small amount of code into the core cluster (i.e., the cluster that contains the core code) for managing code loading at runtime. We call this code as *overlay manager*. The overlay manager consists of:

- A *dynamic loader*, which is passed an address that is the target of a control transfer instruction into the code that needs to be loaded into the memory. The dynamic loader looks up this address in the cluster address table[1] to identify the cluster that it belongs to, then loads the code for that cluster from secondary storage into the code buffer.
- Two *control transfer routines*: `_dynamic_call` and `_dynamic_jmp`. The first of these, `_dynamic_call`, handles the case where control transfer into the target cluster is a function call, while the second routine, `_dynamic_jmp`, handles the case where the control transfer is a jump instruction. Conceptually, these two routines are very similar in their essential functionality: they invoke the dynamic loader to load the target cluster into memory, translate the target address into the appropriate offset within the code buffer, then branch to that location. The only difference between them is that when the control transfer is a function call, the return from that call continues execution at the instruction after the call instruction, and some extra book-keeping is necessary to handle this, as described below.

The inter-cluster control flow edges where the target cluster is not (or, in the case of indirect control transfer, may not be) the core cluster need to be changed so that the overlay manager can take over the control of the execution. The transformation is rather straightforward. We transform the code to push the target address

---

[1] The cluster address table stores the starting address of each dynamically loaded clusters(since dynamically loaded clusters are placed in contiguous address space, it is enough to keep only the starting address of each dynamically loaded cluster in the table). The table is loaded into the memory at the very beginning when kernel starts.



(a) at the point of a call to f

(b) at the point of return from f

Figure 3: The problem with function calls across dynamically loaded clusters

of inter-cluster control flow edges on the stack and then branch to the appropriate control transfer routines:

- A direct unconditional jump ‘jmp ℓ’ is transformed to code that simply pushes the target address and jumps to _dynamic_jmp:

```
push ℓ
jmp  _dynamic_jmp()
```

- The code for an indirect jump ‘jmp *r’ is similar as a direct jump:

```
push *r
jmp  _dynamic_jmp()
```

- A conditional jump instruction ‘J_{cc} ℓ’ is transformed to code of the form

```
     Jcc  A
     ...

A: push ℓ
     jmp  _dynamic_jmp()
```

The transformation for function calls is analogous to that shown above for unconditional jumps and indirect jump, except that it branches to _dynamic_call. Since the transformation makes changes to the stack, the control transfer routines need to clean up the stack before it branches to the code buffer.

The control transfer routines _dynamic_call and _dynamic_jmp are very similar except that there is one important difference between these routines that arises from a subtlety in dealing with function calls from one dynamically loaded cluster into another. This is illustrated in Figure 3. Suppose we have a function call from a dynamically loaded cluster A to a function f in a different dynamically loaded cluster B. Figure 3(a) shows the machine state at the point of the call: the code buffer contains cluster A, and the return address, at some offset m in the code buffer, is the instruction following the call instruction. Since cluster B is also dynamically loaded, this call causes the code for B to be loaded into the code buffer, thereby overwriting the code for A that had been there. When control returns from the function f, therefore, the return address—which is simply the address of offset m in the code buffer—actually points to an instruction in cluster B rather than one in cluster A.[2]

We deal with this problem as follows. The control transfer routine _dynamic_call checks to see whether the return address is within the code buffer. If it is, it creates a restore stub routine dynamically for the return address. The purpose of the restore stub routine is to reload the cluster A from which the call originated, then jump to the appropriate offset within the code buffer. The return address passed to the callee is modified to point to the restore stub. The restore stub simply invokes the dynamic loader with the appropriate address to cause the cluster for the originating call to be loaded,[3] then cleans up the stack and branches to the location within the code buffer corresponding to the instruction following the original call instruction.

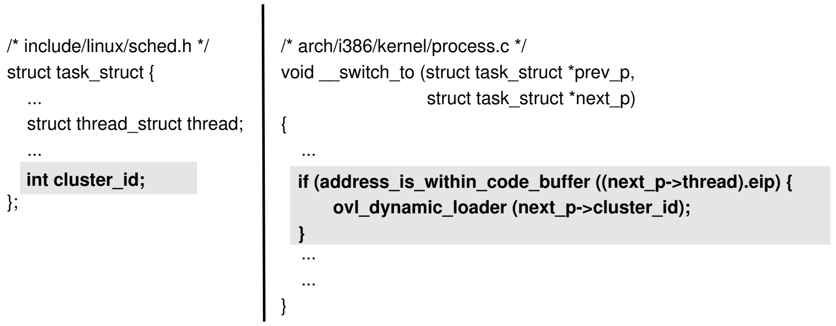
The mechanism is able to handle chains of calls among different clusters properly as well. For example, suppose that there is a call chains, $a \rightarrow b \rightarrow c \rightarrow d$, where a, b, c, and d are functions belonging to different clusters. There are 3 different restore stubs created for this call chain–one for each call-site. When a function, say d, returns, the program control first jumps to the corresponding restore stub for call $c \rightarrow d$ and the restore stub calls dynamic loader to load the cluster of which c belongs to into the code buffer. Then the program counter is set at the appropriate location in the code buffer so that the execution can be continued in function c. The other two restore stubs act in a similar way when the function c and b returns.

The size of a restore stub is very small and consumes little amount of memory. For efficient purpose, a multiple instances of restore stubs are created initially and during runtime, the number of restore stubs will be increased if necessary. In practice, we choose to first create 20 restore stubs and reuse them. Because the Linux kernel has a small fixed-size stack (only 4KB in our tested kernel), the call stack of the Linux kernel is usually not very deep. This number is large enough for all the experiments we tested.

---

[2] In an architecture with variable-length instructions, such as the Intel x86, the return address may not even refer to a valid instruction.

[3] Since the return address is actually within the code buffer in this case, the routine _dynamic_call maps it back to an address that, when passed to the dynamic loader by the restore stub, causes it to load the appropriate cluster into memory.



```
/* include/linux/sched.h */
struct task_struct {
   ...
   struct thread_struct thread;
   ...
   int cluster_id;
};
```

```
/* arch/i386/kernel/process.c */
void __switch_to (struct task_struct *prev_p,
                          struct task_struct *next_p)
{
   ...
   if (address_is_within_code_buffer ((next_p->thread).eip) {
         ovl_dynamic_loader (next_p->cluster_id);
   }
   ...
   ...
}
```

Figure 4: Source code changes to handle multi-threading

3.4 Context Switches and Interrupts

There are two issues that have not been addressed in the discussion so far. The first is that of multi-threading, which is a feature typical of modern general-purpose operating system kernels. The second is that of control transfers due to interrupts.

We extend our approach to handle multi-threading via minor (manual) modifications to the kernel code, as follows. We add a single new field to the thread state (in the Linux kernel, the structure `task_struct`) to identify the cluster whose code is being executed by that thread. The additional memory requirements for this are small, just 4 bytes per thread. This field is initialized and updated by the dynamic loader as needed during execution. The code for the scheduler is modified to check the program counter value for a thread that is about to run, and to invoke the dynamic loader to load the appropriate cluster into the code buffer if this address is within the code buffer. The code changes necessary are shown in Figure 4, where the additional code that has to be introduced in our approach has been highlighted.

One issue we have not discussed is that of dealing with interrupts. The problem of dealing with interrupt in the kernel is, when the kernel is executing a code cluster A in the code buffer and an interrupt happens, if the the interrupt handler brings a different code cluster B into the code buffer, B will overwrite cluster A in the code buffer. Once the interrupt handler finishes and the kernel execution returns back to its previous normal execution, if A is not reloaded into the code buffer, an error will happen. In our current implementation, we handle this is issue by making sure that interrupt handlers are part of the core code and therefore always remain in memory so that interrupt handler will not load new code cluster into the code buffer. This, however, will obviously increase the size of core code. There are other ways to handle this. One approach is to have two code buffers: one is dedicated to the normal kernel execution other than interrupt handling; while the other is dedicated only to the interrupt handling. Another approach is to modify the return process of interrupt handler so that the required code cluster is loaded into the code buffer before the interrupt handler returns. We are currently working on investigating and evaluating these different methods.

4. EXPERIMENTAL RESULTS

We have implemented our ideas using the PLTO binary rewriting system for the Intel x86 architecture [14] and evaluated them using version 2.4.31 of the Linux kernel. PLTO is also used to collect profiles for the kernel. An OS kernel like Linux kernel, does not have well-defined entry and exit points as the ordinary application to server as natural points to start and end profiling. Therefore, our system uses a special(new) system call to begin profiling as well as ending profiling and write out profile data. More details of our profiling mechanism is discussed in [13]. In our current implementation, the dynamically loaded code is still stored in memory, but in a separate section in the kernel binary. The code is loaded into the code buffer through `memcpy` function call. We plan to change the implementation to load code from secondary storage once we setup a developing environment that can simulate an embedded system integrated with flash memory.

To get an accurate evaluation of the efficacy of this system, we began with a minimally configured kernel where as much unnecessary code as possible has been eliminated by configuring the kernel carefully. For our experiments, we therefore configured the Linux kernel to remove modules, such as the sound card and video support, that are not required to run our benchmarks. We considered two versions for the kernel: one with networking support, the other without. The kernel code was compiled with *gcc* version 3.4.4 using the compilation flags of '`-Os`', which instructs the compiler to optimize for code size. The code sizes for the resulting kernels (only the `.text` sections) are as follows:

Kernel	*Code size* (bytes)
2.4.31, w/o networking	590,022
2.4.31, with networking	890,793

In order to simplify the booting process of the Linux kernel, we modified the kernel boot up file `inittab` so that the Linux kernel will run in single user mode (level 1). All the experiments are conducted using a Intel Pentium 4 3 GHz desktop machine with 2GB memory installed.

We used three sets of benchmarks to evaluate our ideas: MiBench [7], a widely used and freely available collection of benchmark programs for embedded systems; MediaBench, a suite of programs used for evaluating multimedia and communications systems [9]; and *httpd*, the Apache HTTP server (version 2.0.50), which was used because it exercises more of the kernel code, and in different ways, than the programs in the MiBench and MediaBench suites. Our experiments with MiBench and MediaBench used the version of the kernel compiled without networking, while *httpd* was tested on the kernel version containing networking.

4.1 Compaction Results

Table 1 shows the behavior of our clustering algorithm and the compaction results for the different benchmarks; the data presented

	Core code size bound (γ) (%)	Cluster Statistics			Compaction Results	
		No. of Clusters	*Ave. cluster size (bytes)*	*Max. cluster size (bytes)*	*Total memory size (bytes)*	*Memory size reduction (%)*
MiBench	0	250	1634	2023	255,566	56.7
	2	247	1635	2041	260,068	55.9
	4	244	1634	2025	264,639	55.1
	6	241	1635	2032	269,288	54.4
	8	238	1636	2039	273,748	53.6
	10	235	1638	2045	277,988	52.9
MediaBench	0	250	1639	2045	254,573	56.9
	2	248	1632	2025	259,041	56.1
	4	245	1633	2022	263,629	55.3
	6	242	1633	2025	268,332	54.5
	8	239	1634	2026	272,762	53.8
	10	236	1635	2000	277,073	53.0
httpd	0	385	1650	2039	368,600	58.6
	2	381	1649	2042	374,948	57.9
	4	377	1650	2043	381,289	57.2
	6	373	1650	2044	387,557	56.5
	8	368	1654	2042	394,121	55.8
	10	364	1653	2045	400,620	55.0

Table 1: Clustering statistics and compaction results for different core code size bounds with code buffer size = 2 KB

corresponds to a code buffer size of 2 KB. This value was chosen because it is the page size on flash memory chip considered in Section 4.2.

The first column indicates the benchmark suite being considered. The second column gives the core code size bound γ indicating how much the core code is allowed to grow in size. The third column gives the number of clusters formed. The fourth and fifth columns give the average and maximum cluster size, respectively. The sixth column gives the total memory size, computed as the sum of the sizes of the memory-resident code, the code buffer and the memory allocated by overlay manager, which includes the restore stubs (600 bytes) and the cluster address table (= no. of clusters $\times$ 4 bytes). The size of the memory-resident code is obtained as the size of the `.text` section in the compacted kernel.[4] The final column gives the percentage reduction size, measured relative to the size of the original kernel (Recall that the MiBench and MediaBench programs were evaluated on a kernel without networking support, with original size 590,022 bytes, while the *httpd* benchmark was evaluated on a kernel with networking support, with original size 890,793 bytes).

It can be seen from Table 1 that, as expected, increasing the value of the code size bound γ leads to a decrease in the total number of clusters. The average cluster size remains almost the same , while the maximum cluster size varies for all different γ. Since the code buffer size is being held constant in our experiments (2 KB), the total memory size reduction achieved decreases as γ—and therefore the amount of memory-resident code—increases. The memory size reductions achieved are fairly consistent across our benchmarks, and range from about 56%–58% for $\gamma = 0$ to about 53%–55% for $\gamma = 0.1$. The next section examines the effect of different values of γ on the runtime cost of code loading.

4.2 Cost of Code Loading

Table 2 shows the effect of different core code growth bounds on the runtime cost of on-demand code loading. We show data for two different costs: the first set of data (columns 3–5) show the cost of booting the kernel and starting a shell(for running *httpd*, the booting process also includes starting network and httpd server); while the second set (columns 6–8) show the kernel-level cost of running the benchmark applications. Columns 3 and 6 give the total number of accesses for code loading while columns 4 and 7 give the total amount of code loaded into the code buffer. The application codes were run as follows: for the MiBench suite, we ran both the small and large input sets that came as part of the suite; for MediaBench, we used the run scripts provided with the benchmark applications; for *httpd*, we used the command [5]

```
ab -n 5000 -c 2 http://test_addr
```

which sends a total of 5000 requests, 2 at a time, to the test machine whose IP address is given by *test_addr*. For both sets of data, booting the kernel and running the benchmark programs, the number of accesses for code loading and the total amount of code loaded into code buffer decrease as the code growth bound γ is increased.

Since our experiments were done on a relatively fast desktop environment, the small amount of time spent in the operating system kernel, together with the granularity of the system clock, made it difficult to reliably measure the effect of our dynamic code loading scheme on the total time spent within the kernel. Instead, we give a rough estimate of the effect of such a scheme in an embedded context.

First, we estimate the time taken for dynamic code loading out of flash memory secondary storage using manufacturer's data sheets for a typical commercial flash memory currently in use. For this, we (quite arbitrarily) chose the Micron MT29f2G08AAb NAND flash memory [10]. This is a 2 GB flash memory unit where data is stored in 2 KB pages. Data reads are done a page at a time (i.e., the smallest unit of data read is 2 KB), and it takes 130.9 microseconds to read each page. We calculated the estimated cost of code loading

---

[4] There is some code in the `.init.text` section used during the kernel bootup process, but we did not include this in our size computations because this section is deallocated, and its memory freed up, after the initial portion of booting.

[5] `ab` is the Apache HTTP server benchmarking tool.

	Core code size bound (γ) (%)	Kernel boot data			Application execution data		
		No. of accesses	*Total code loaded* (KB)	*Est. load cost* (sec)	*No. of accesses*	*Total code loaded* (KB)	*Est. load cost* (sec)
MiBench	0	66,736	111,842	8.68	811,218	1,362,556	105.46
	2	43,933	73,817	5.71	226,395	387,725	29.43
	4	16,964	28,537	2.21	7,939	13,343	1.03
	6	8,647	14,080	1.12	3,131	5,016	0.41
	8	3,412	5,644	0.44	2,326	3,871	0.30
	10	1,700	2,787	0.22	1,091	1,804	0.14
MediaBench	0	65,964	109,222	8.58	40,109	67,016	5.21
	2	27,802	46,825	3.61	17,992	30,100	2.34
	4	11,389	19,037	1.48	4,131	7,015	0.54
	6	5,831	9,472	0.76	2,532	4,244	0.33
	8	3,130	5,009	0.41	929	1,536	0.12
	10	1,723	2,861	0.22	646	1,069	0.08
httpd	0	96,111	163,719	12.49	162,027	261,364	21.06
	2	23,735	39,375	3.09	51,229	78,517	6.66
	4	11,078	18,326	1.44	10,451	17,194	1.36
	6	2,861	4,657	0.37	341	567	0.04
	8	1,620	2,708	0.21	529	906	0.07
	10	1,030	1,696	0.13	405	688	0.05

Table 2: Runtime cost of dynamic code loading for different core code growth bounds

as

$$Est.Cost = \sum_i \lceil \frac{size(i)}{2048} \rceil \times access(i) \times 130.9\mu s,$$

where $size(i)$ is the size of cluster i and $access(i)$ is the total number of times of which cluster i was loaded into code buffer. The estimated cost of code loading is shown in columns 5 and 8 in Table 2.

Secondly, we tried to evaluate the impact of dynamic code loading on the performance of the application programs running on the kernel. We estimate this by considering the time taken to run each of the three benchmarks on an unmodified kernel (i.e., the cost of code loading is zero). On average, the total time for running each benchmark on an unmodified kernel is as follows:

Benchmark	*Running time* (sec)
MiBench	18.82
MediaBench	3.53
httpd	3.82

Using MiBench as an example, what the data shown is that, if we were to use dynamic code loading on our desktop environment, using the flash memory described above, choosing $\gamma = 0$ would yield a 56.7% reduction in code size, but would lead to an execution time of $18.82 + 105.46 = 124.28$ secs, i.e., almost 7 times of the runtime on an unmodified kernel. On the other hand, for $\gamma = 10\%$, we see a 52.9% reduction in code size while the total runtime goes to $18.82 + 0.14 = 18.96$ seconds, an increase smaller than 1%. Figure 5 shows that for all three benchmarks, the overhead of dynamic code loading reduces significantly when code growth bound is increased from 0% to 4%. The reason for this dramatic performance improvement is not hard to see: if the frequently executed parts of the kernel is kept in memory, they will not have to be loaded repeatedly from secondary storage.

It is important to note that these numbers are a conservative upper bound on the runtime overhead that would actually be incurred on an embedded platform. The embedded platforms our technique is aimed at are likely to be considerably slower than the desktop used for our experiments, which means that the time taken to run the whole MiBench would be correspondingly much greater than the 18.82 seconds used for these calculations. Since the flash memory characteristics remain the same, it seems reasonable to conclude that the actual runtime overheads experienced on an actual embedded system would be even lower.

5. RELATED WORK

Code compaction of operating system kernels has been considered by Chanet *et al.* [3, 2] and He *et al.* [8]. They apply traditional size-reducing compiler optimizations to eliminate dead, unreachable, and duplicate code [3, 8] and compression of rarely executed code with on-demand decompression [2]. These works keep all of the compacted kernel in memory, which limits the extent of memory footprint reduction they are able to achieve.

There has been a great deal of other work on automatic code compaction (see the survey by Beszédes *et al.* [1]). Most of this work focuses on application code and does not address the complications that arise in dealing with operating system kernels.

We are not aware of a great deal of other work on binary rewriting of operating systems kernels. Flowers *et al.* describe the use of Spike, a binary optimizer for the Compaq Alpha, to optimize the Unix kernel, focusing in particular on profile-guided code layout [6]. This work focuses on improving execution speed rather than reducing code size and therefore uses techniques very different from ours.

Recently, some researchers have begun exploring the use of overlays out of flash memory to reduce memory requirements in embedded systems. Park *et al.* describe a scheme for generating dynamic code overlays for programs that can be modeled using synchronous data flow, which makes it possible to determine a static schedule for the program's code [12]. Park *et al.* describe an application-specific demand paging mechanism for low-end embedded systems that do not have virtual memory [11]. Both works limit their focus to application programs and do not address the numerous issues peculiar to operating system kernels that arise in this context.

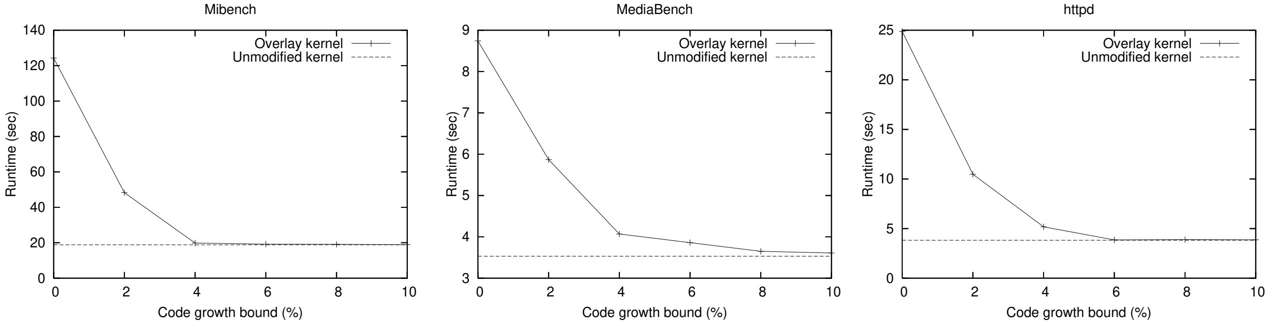


Figure 5: The estimated runtime cost of kernel with overlay comparing with unmodified kernel

Egger *et al.* describe dynamic code placement techniques and memory management strategies for scratchpad memory in embedded systems [4] [5]. Their interest focuses on improving the overall performance of system instead of reducing the memory requirement. They also apply their work only on application programs.

6. CONCLUSIONS

Recent years have seen an increasing trend towards the use of general-purpose operating systems, such as Linux, in embedded systems. This solution, however, this has the disadvantage that general-purpose OS kernels, by their very nature, contain a lot of code that is not needed in an embedded context. This is a problem because embedded devices typically have a limited amount of memory available. This paper describes an approach to reducing the memory requirements of the OS kernel using an on-demand code overlay mechanism. Our approach is based on a post-link-time binary rewriter that uses edge profile information to carry out code clustering. Experiments with the Linux kernel show that we are able to reduce the memory requirements of the kernel code to 53% with little degradation in performance.

7. ACKNOWLEDGMENTS

We would like to thank Gernot Heiser and the anonymous reviewers for their helpful comments on drafts of this paper and Somu Perinayagam for his work on the binary rewriter tool.

8. REFERENCES

[1] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, 2003.

[2] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Automated reduction of the memory footprint of the linux kernel. *ACM Transactions on Embedded Computing Systems*. To appear.

[3] 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. 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 95–104, June 2005.

[4] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233, New York, NY, USA, 2006. ACM Press.

[5] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330, New York, NY, USA, 2006. ACM Press.

[6] R. Flower, C.-K. Luk, R. Muth, H. Patil, J. Shakshober, R. Cohn, and P. G. Lowney. Kernel optimizations and prefetch with the Spike executable optimizer. In *Proc. 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[7] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. MiBench: A free, commercially representative embedded benchmark suite. pages 3–14, December 2001.

[8] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews. Code compaction of an operating system kernel. In *Proc. Fifth International Symposium on Code Generation and Optimization (CGO)*, pages 283–295, March 2007.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th IEEE International Symposium on Microarchitecture (Micro '97)*, pages 330–335, December 1997.

[10] Micron Technology. Small block vs. large block NAND devices. Technical Report Technical Note TN-29-07 (Rev. B), February 2006.

[11] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 114–124, New York, NY, USA, 2004.

[12] H. Park, K. Oh, S. Park, M. Sim, and S. Ha. Dynamic code overlay of sdf-modeled programs on low-end embedded systems. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 945–946, 2006.

[13] M. Rajagopalan, S. Perinayagam, H. He, G. Andrews, and S. Debray. Binary rewriting of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, October 2006.

[14] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.