Quasi-Static Shared Libraries and XIP for Memory Footprint Reduction in MMU-less Embedded Systems*

Jiyong Park, Jaesoo Lee, Saehwa Kim, and Seongsoo Hong

Real-Time Operating Systems Laboratory, School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea
{parkjy, jslee, ksaehwa, sshong}@redwood.snu.ac.kr

Abstract

Despite a rapid decrease in the price of solid state memory devices, system memory is still a very precious resource in embedded systems. The use of shared libraries and XIP (eXecution-In-Place) is known to be effective in significantly reducing memory usage. Unfortunately, many resource-constrained embedded systems lack an MMU, making it extremely difficult to support these techniques. To address this problem, we propose a novel shared library technique called a quasi-static shared library and an XIP, both based on our enhanced position independent code technique. In our quasi-static shared libraries, global symbols are bound to pseudo-addresses at linking time and actual physical addresses are bound at loading time. Unlike conventional shared libraries, they do not require symbol tables that take up valuable memory space and, therefore, allow for expedited address translation at runtime. Our XIP technique is facilitated by our enhanced position independent code where a data section can be arbitrarily located. Both the shared library and XIP techniques are made possible by emulating an MMU's memory mapping feature with a Data Section Base Register (DSBR) and a Data Section Base Table (DSBT).

We have implemented these proposed techniques in a commercial ADSL (Asymmetric Digital Subscriber Line) home network gateway equipped with an MMU-less ARM7TDMI processor core, 2-MB flash memory, and 16-MB RAM. We measured its memory usage and evaluated its performance overhead by conducting a series of experiments. These experiments clearly demonstrate the effectiveness of our techniques in reducing memory usage. The results are impressive: 35% reduction in flash memory usage when using only the shared library and 30% reduction in RAM usage when using the shared library and XIP together. These results were achieved with only a negligible performance penalty of less than 4%. Even though these techniques were applied to uClinux-based embedded systems, they can be used for any MMU-less real-time operating system.

Keywords

Shared library, quasi-static linking, XIP, MMU-less, embedded systems, memory footprint reduction

* This paper is an extended version of the paper that appeared in the Proceedings of 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006) [1].

1. Introduction

Embedded systems are generally characterized as systems which have stringent constraints on resources such as processing power, energy and memory size. In spite of recent remarkable advances in semiconductor technology, memory size in particular is still quite limited. In general, larger memory occupies a larger physical space, consumes more power and costs more. Consequently, utilizing memory efficiently is still a driving factor that affects all aspects of system design. Currently, many embedded systems are constructed from a combination of COTS (Commercial Off-The-Shelf) software components like Linux-based operating systems [2, 3] and the Busybox multi-call utility [4]. Since such COTS software components are already heavily pre-optimized, it is technically very hard for developers to further reduce memory usage of the components to make them to fit into systems with tight memory constraints. Furthermore, this requires sometimes unexpected and often error-prone tasks like optimizing program code at the instruction level by hand. For example, even these days, embedded systems developers are sometimes forced to utilize surprisingly old memory management techniques such as overlays, which make program development complicated. In this paper, we present two effective techniques, a quasi-static shared library and an eXecution-In-Place (XIP), to reduce memory usage in MMU-less embedded systems. Also, we present the results of applying these techniques to a commercial ADSL (Asymmetric Digital Subscriber Line) [5] home network gateway.

A shared library [6-11], in general terms, is a library that contains routines that are loaded into memory by an operating system as needed and dynamically shared with other applications without static linking. If a shared library is used, an application need not contain any routines present in that library. Instead, it can share a single copy of those routines which is already loaded into memory with other applications. This helps effectively reduce the amount of space used in a file system and runtime memory, and allows shared library routines to be replaced on-the-fly. An MMU (memory management unit) is usually required to support the shared library technique. A shared library is composed of a public text section that is shared among processes and a private data section that is separately allocated for each process. Therefore, when a library routine accesses a data symbol, it must be redirected to the private data section of the executing process. In an MMU-equipped system, this is easily achieved by mapping the data section of a library to a fixed area in the virtual address space of a process. The library routine then uses the virtual address of a data symbol. However, an MMU is rarely used in resource-constrained embedded systems since it greatly increases both the production cost and complexity of a system. Thus, shared libraries are not commonly implemented in embedded systems as compared to servers or desktop PCs that contain an MMU.

Before we consider the implementations of shared library frameworks on MMU-less embedded systems, we first make a distinction between two types of shared libraries based on the time global symbols are bound to actual addresses. One type is a dynamically linked shared library [6, 11] in which symbols are bound to addresses at loading time. The other is a statically linked shared library [6] in which symbols are bound to addresses at linking time. The shared library framework implemented by Cadenux [12] falls under the category of dynamically linked shared libraries. In this kind of framework, a dynamic loader fixes all references to a shared library in an application using a symbol table contained in the shared library. This means that a symbol table has to be maintained throughout the lifetime of a shared library, taking up valuable memory space. This also causes a substantial runtime performance penalty due to the time-consuming dynamic linking performed when a process accesses a library symbol for the first time. Hence, dynamically linked shared libraries are not appropriate for embedded systems in which memory and CPU resources are restricted.

Unfortunately, statically linked shared libraries are not appropriate for MMU-less embedded systems either, since each of the libraries in a system must be allocated to a non-overlapping region in a single address space. This needs to be done to avoid conflicts among different libraries but has the potential effect of incurring serious memory waste in MMU-less embedded systems.

To gain the advantages of both dynamically and statically linked shared libraries while avoiding the aforementioned problems, we propose a quasi-static shared library. This is a library that is loaded at an arbitrary address but statically linked to an application. Since the quasi-static shared library does not require a symbol table, we can save memory and expedite the process of computing the actual addresses of symbols.

XIP [13, 14] is also very effective in reducing RAM usage. XIP is defined as a technique whereby a program stored in ROM or flash memory is run directly from the location where it is stored. As a result, the text section of a program or a library does not have to be copied into RAM. Like shared libraries, however, XIP requires support from an MMU since a data section of an XIP process is relocated to the RAM area and memory references to the data section have to be relocated accordingly. In an MMU-less embedded system, this requires patching global symbol references in a text section that is stored in a read-only file system.

A straightforward way to avoid patching a text section for XIP is to place a data section at a fixed location reserved for each program. However, this leads to a significant waste in memory usage in an MMU-less embedded system. Thus, we emulate an MMU by applying similar techniques used in our quasi-static shared library and show that our approach yields an efficient implementation of XIP without an MMU.

We have implemented the proposed shared library and XIP techniques in a commercial ADSL home network gateway. This system is equipped with an MMU-less ARM7TDMI processor core, 2-MB flash memory and 16-MB SDRAM. A derivative of Linux for microcontrollers without an MMU, uClinux [2], is used as the operating system. The target system is very complex, running nearly 18 out of 52 applications simultaneously during the course of its usual operation. The implementation of our proposed techniques includes modification and optimization of the compiler, linker, uClinux's loader and CRAMFS (Compressed RAM File-System). We applied our shared library and XIP techniques to the 52 applications and one library. We have performed a series of experiments to measure memory usage and evaluate performance overhead. With only the quasi-static shared library technique applied, flash memory usage is reduced by 35% from 1173 KB to 768 KB with a small decrease in RAM usage. On the other hand, if both the shared library and XIP techniques are applied together, the worst case RAM usage is reduced by 30%. The execution time of an application is affected minimally by less than 4% in both cases.

The rest of the paper is organized as follows. In the next section, the target system's original linking and loading process is explained. Section 3 presents the mechanisms behind our quasi-static shared library and XIP techniques. Section 4 explains how these mechanisms are coherently integrated into an overall system. Section 5 provides the results of an empirical evaluation of our techniques in the target system. Section 6 describes related work and Section 7 serves as our conclusion.

Table I. Hardware and software components of the target system.

Hardware	Main processor	S5N8947 (ARM7/TDMI core, No MMU, 40 MHz)
	ADSL processor	S5N8950 ADSL DSP, S5N8951 AFE
	Memory	16-MB SDRAM, 2-MB NOR Flash memory
	Interfaces	ADSL × 1, Ethernet × 1, RS-232C × 1
Software	Bootloader	ARMboot 1.0.2
	Operating system	uClinux 2.4.17 (Linux for MMU-less processor)
	File-systems	CRAMFS (mounted on flash memory, program storage, read-only)
	Application programs	shell, web server, NAT, firewall, IP filtering, DHCP server, SNMP server, system configuration CGI programs, and etc. (total 52)
	Libraries	uClibc 0.9.15 (standard C library)
	Development tools	gcc 3.2.1 (compiler), binutils 2.13.1 (linker, assemblers, and etc.), and elf2flt (conversion tool from ELF to BFLT)

2. Target System Components and Conventional Linking

To aid in understanding the rest of this paper, we show the target system's hardware and software components, and explain the original process of linking, and loading executable code and the organization of the file system image. Table I shows the components of the target system. Basically, it is a modem that is connected to the Internet through ADSL and provides Internet connectivity for other home network devices.

The original system was constructed with conventional static libraries where library routines are statically linked and copied to applications. When the executable code image of an application is created, symbol addresses are determined with an assumption that executable code will be loaded from address `0`, as in systems with an MMU. Where a target system lacks an MMU, it is impossible to provide all processes with separate virtual address spaces starting from address `0`. Thus, executable code images must be relocated to arbitrary starting addresses. This requires a process to contain a data structure that enables the modification of symbol addresses that it uses. It is called a relocation table and is used by a loader when an executable code image is loaded into memory.

In the target system, all created executable code images are collected together to create a CRAMFS image. The flash memory contains this CRAMFS image as well as a bootloader and a compressed image of uClinux. The executable code images in CRAMFS may be either compressed or uncompressed as desired by the developer. In practice, most are compressed.

An executable code image stored in flash memory is loaded into RAM by uClinux's loader. At loading time, it calculates the symbols' absolute addresses and modifies the text and data sections using the relocation table.

3. Design Requirements and Solution Mechanisms

As mentioned in previous sections, our objective is to implement shared library and XIP mechanisms without support from an MMU. Furthermore, we seek to reduce time and space overhead by avoiding dynamic data structures, such as a symbol table, while maintaining compatibility with conventional static libraries so that existing applications need not be rewritten. To meet these design goals, it is necessary to modify the code generator of the `gcc` compiler [15], uClinux's loader and the CRAMFS. We also have to implement a tool for creating pseudo-libraries that play the role of symbol tables used in conventional shared libraries.

In this section, we first enumerate design requirements for quasi-static shared libraries and XIP in Section 3.1 and 3.2, respectively. After that, we describe our specific mechanisms for satisfying these requirements in each subsection. Subsequently, in Section 4, we show how these mechanisms are coherently integrated into the overall system.

3.1 Quasi-Static Shared Libraries

In designing our shared library framework for memory-constrained MMU-less embedded systems, we have kept the following three design requirements in mind.

CS1. The shared libraries should be able to operate regardless of their location in memory, and their text sections should not have to be modified once they are loaded. (A shared library created with a specific, statically determined load address in RAM causes a significant waste in memory usage: the memory space pre-assigned to a library cannot be used for any other purpose even though the library is not actually used.)

CS2. A single process should be able to use multiple shared libraries. (Without this functionality, our shared library technique would be very restrictive to developers.)

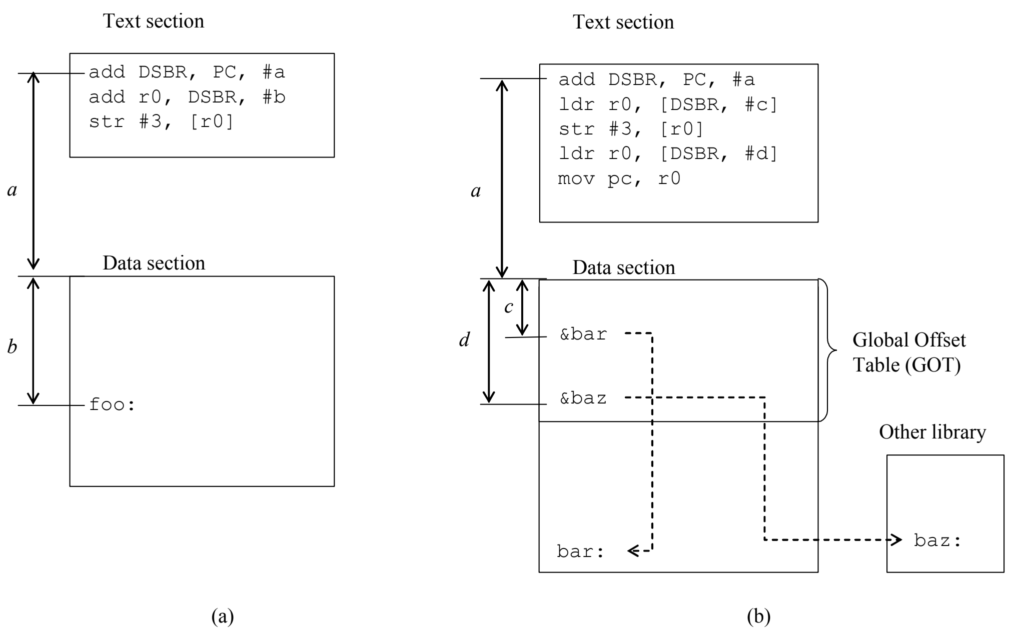
CS3. The loader should be able to operate without a symbol table. (We therefore avoid incurring the associated performance penalties and space overhead.)

To meet these requirements, a variety of techniques are employed. For requirement CS1, we rely on a data section base register (DSBR) and a global offset table (GOT) [6], as well as a form of position independent code that we enhance specially for MMU-less systems. To meet requirement CS2, we have devised the data section base table (DSBT) for storing the base addresses of multiple data sections for all loaded shared libraries in a process. Finally, to satisfy requirement CS3, we have developed quasi-static linking.

3.1.1 Position Independent Code Enhanced for Requirement CS1

It is well known that position independent code (PIC) [6, 11] can be executed from any location in memory without requiring modification of a text section. This is possible because PIC does not use an absolute address when referring to a symbol. In fact, the PIC technique is used in most shared libraries, and many widely available compilers including `gcc` are able to generate PIC. Unfortunately, without an MMU, it is very difficult, though not impossible, to utilize PIC in implementing shared libraries since the PIC technique is based on the assumption that a data section is located at a known distance from a text section. In order to overcome this difficulty, we enhance the PIC technique. Before presenting our enhancement, we first explain the addressing mechanism used in the conventional PIC technique.

In PIC, symbols can be generally classified into three categories depending on how their addresses are dealt with: (1) local static functions, (2) local static variables and (3) global variables and functions. Referencing local static functions is



```
add DSBR, PC, #a
add r0, DSBR, #b
str #3, [r0]

foo:
```

```
add DSBR, PC, #a
ldr r0, [DSBR, #c]
str #3, [r0]
ldr r0, [DSBR, #d]
mov pc, r0

&bar
&baz
bar:
baz:
```

Fig. 1. Pseudo code for referencing symbols in MMU-equipped systems: (a) writing an integer constant to local static variable `foo` using PC-relative addressing and (b) writing an integer constant to global variable `bar` defined in the same library and then jumping to global function `baz` defined in the other library through GOT. Note that `&bar` and `&baz` in GOT represent the absolute addresses of `bar` and `baz`, respectively.

fairly insignificant since they are contained within the same text section as the calling function. This is easily done through PC-relative addressing. In contrast, referencing local static data naturally involves locating the data section. Usually, in MMU-equipped systems, the data section of a routine is located at a known distance from the text section as shown in Fig. 1 (a). Since this distance, denoted *a* in Fig. 1 (a), is fixed at linking time, the data section can be located in a PC-relative manner. In this way, the base address of the data section is loaded into a register called the data section base register (DSBR). Individual symbols for variables can then be located by adding an offset, denoted *b* in Fig. 1 (a), to the DSBR. On the other hand, referencing global variables and functions is done through a global offset table (GOT) as shown in Fig. 1 (b). The GOT is a table that contains entries for all global symbols referenced in PIC. The absolute address of a global symbol is determined and recorded by a loader at loading time. In order to refer to a global symbol later on, a compiler generates code that accesses the entry for that symbol in the GOT and fetching the absolute address recorded in that entry. Pseudo code for doing this is given in the text section in Fig. 1 (b). The GOT is located at a fixed offset in the data section, usually at the beginning. Thus, once the data section of a library is located, both global symbols and local variables can be referenced. In MMU-equipped systems, a compiler is responsible for generating an instruction to load the DSBR with the data section's base address.

Unlike addressing techniques explained above, in an MMU-less system, the data section of a library, and thus its GOT, are located at an unknown distance away from the text section. Therefore, PC-relative addressing cannot be used to load DSBR. In our technique, the DSBR is managed differently by a loader and a compiler. Specifically, it is initialized by the operating system's loader when a process is loaded for execution. Since the loader determines the data section base addresses of shared libraries used in a process, it can provide them for the process.

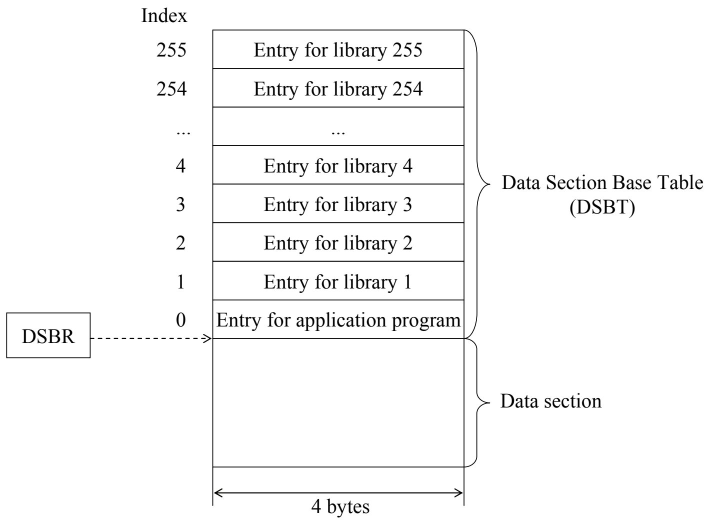


Fig. 2. Structure of DSBT and method for accessing its entries. For example, the entry indexed 3 in which that the address of data section of library 3 is stored, there are 16 (= 4 × 3 + 4) bytes before the start of data section pointed by DSBR.

Once the DSBR is initialized, application programs or shared libraries should not overwrite the register. This requires that application programs and shared libraries be compiled with an option that exclusively reserves a general purpose register as a DSBR. Clearly, this may prevent binary-only libraries from being used in our approach since they are not aware of the DSBR. In order to solve this problem, we carefully select one of callee-saved registers as a DSBR. This guarantees that the value of the register remains the same before and after a function call to a binary-only library. As a result, binary-only libraries can be still statically linked with application programs or other libraries in a conventional manner. As a matter of fact, such a binary-only library cannot take advantage of our technique. For the ARM processor, which is our target processor, we use the `sl` register as a DSBR and this register is specified as one of callee-saved registers [45].

Note that the loss of one general-purpose register does not cause any noticeable impact on program execution performance. This can be confirmed via experimental results. For processors with 32 or more registers, experimental results show that the performance degradation is negligible [43][44]. For the ARM processor, which has only twelve general-purpose registers, the impact is also measured to be small in our experiments. The experimental results are given in Section 5.

3.1.2 Data Section Base Table for Requirement CS2

In a process that uses multiple libraries, there are multiple text and data sections each of which is loaded at an arbitrary address. Thus, the DSBR of a process must point to different data sections as application-to-library or inter-library calls are invoked. To do so, we devise a runtime data structure named a data section base table (DSBT). The DSBT is a table that contains the data section base addresses of the libraries that are loaded for a process.

Fig. 2 shows the structure of the DSBT. By default, it has 256 entries and each entry is four bytes long. Each entry is dedicated to a specific shared library and contains the data section base address of the shared library. In order to index the table, each shared library is given a unique numerical ID, which ranges from 1 to 255. The ID 0 is reserved for the application program itself.

When an application program is loaded into memory for execution, the operating system loader allocates the text section and data section, among others, of the application program. It also allocates the data sections of all shared libraries used in the application program. Then, it creates the DSBT by filling up its entries with the allocated data section base addresses and attaches the DSBT just before the data section of the application program. It goes on replicating the DSBT and attaching its replica before the data section of each shared library. Finally, it sets the DSBR to the data section base address of the application program. We have modified uClinux's loader for this task. Fig. 2 shows the configuration of the DSBR and DSBT right after process loading. Under this technique, an entry corresponding to a shared library with `LIBRARY_ID` can thus be easily located using a simple formula as below.

```
ADDRESS = [DSBR - 4 × LIBRARY_ID - 4]
```

We choose to replicate the DSBT for all shared libraries at the expense of increased memory usage so as to allow a process to access both a data section and the DSBT via a single DSBR, regardless of the shared library that the process is currently executing. In this way, we can avoid using an extra register for accessing the DSBT. Note that general purpose registers are an extremely valuable resource in enhancing the runtime performance of executable code. Moreover, the space overhead is relatively small since the size of the DSBT is configurable according to the actual number of shared libraries used in a system. For example, if seven shared libraries are used in a system, the space overhead of DSBTs is only 224 bytes per process in the worst case: 32 bytes for a DSBT with eight entries and seven DSBTs at maximum. Developers can even merge several distinct libraries at build-time to further reduce the table size.

During execution, when an application-to-library or inter-library call is made, the DSBR must be updated so that it points to the data section base address of the called library. To do so, the process first obtains the library ID of the called library and then computes the address of the DSBT entry corresponding to that library. Since the library ID is statically determined at build-time, the compiler can generate an instruction that computes the address of that DSBT entry. After that, the process saves the original DSBR in the stack and sets the DSBR to the address obtained from the DSBT entry. Conversely, when returning from an application-to-library or inter-library call, the process restores the original DSBR from the stack. We have modified the code generator of `gcc` so that it can produce appropriate instructions in the prologue and epilogue of globally defined library functions.

As a concrete example, below we provide the code of the `printf` function contained in the `uClibc` library. This code is generated by the modified `gcc` and the library ID of `uClibc` is `1`. Recall that the we use `sl` register as a DSBR in the ARM architecture.

```
<printf>:
    (1) : mov      ip, sp
    (2) : stmdb    sp!, {r0, r1, r2, r3}
    (3) : stmdb    sp!, {sl, fp, ip, lr, pc}
    (4) : ldr      r3, [pc, #28]
    (5) : ldr      sl, [sl, -#8]
    (6) : ldr      r3, [sl, r3]
    (7) : sub      fp, ip, #20
    (8) : ldr      r0, [r3]
    (9) : ldr      r1, [fp, #4]
    (10): add      r2, fp, #8
    (11): bl       c28 ; vfprintf
    (12): ldmdb    fp, {sl, fp, sp, pc}
```

The three shaded instructions are those generated by our modified compiler. Instruction `(3)` saves the original DSBR. Instruction `(5)` computes the address of the DSBT entry that corresponds to `uClibc` and sets DSBR accordingly. Instruction `(12)` restores the original DSBR.

Although we demonstrate our technique for the ARM architecture, it is also applicable to other architectures, such as x86, that support multiple segment registers. Even in such architectures, the DSBT is still needed if the number of shared libraries used in a system exceeds the number of available segment registers.

3.1.3 Quasi-Static Linking for Requirement CS3

Finally, to fulfill requirement CS3, we have developed a quasi-static linking mechanism. Contrary to a common intuition, the space and performance overhead caused by symbol tables cannot be underestimated in embedded systems. For example, the size of a symbol table for the `uClibc` library takes up about 29 KB. Quasi-static linking is a novel linking mechanism that is in the middle of the static linking and the dynamic linking. At linking time, addresses of symbols are statically bound to logical numeric addresses. At loading time, the logical numeric addresses are dynamically resolved into the absolute addresses of the symbols using a simple address resolution formula. This permits a shared library to be loaded at an arbitrary address since the permanent address of a symbol is dynamically determined and it does not have to use a symbol table at loading time since symbols are statically bound in any case.

The two core components of the quasi-static linking mechanism are a pseudo-address and a pseudo-library. A pseudo-address for a symbol is defined as a logical numerical address that specifies the library the symbol belongs to and its offset inside the library. A pseudo-address thus consists of two fields: a library ID and an offset as shown in Fig. 3 (a). The library ID and the offset are 8 bits and 24 bits wide, respectively. The offset is defined as the distance from the start of the library to the definition of the symbol, when the data section is placed just after the text section.

A pseudo-library is a library that contains only a symbol table but no actual code. Thus, a pseudo-library cannot be used at runtime to bind library code into an application: a normal shared library object without a symbol table is used instead. Each entry in the symbol table specifies a symbol by its name and its corresponding pseudo-address. As an example, Fig. 4 depicts a symbol table for the `printf.o` object file contained in the pseudo-library of the `uClibc` library (`libc.a`). In this library, the pseudo-address of `printf` is defined as `0x01011ff8`: the function `printf` is defined in a shared library with ID `1`, and its entry address relative to the start of the shared library is `0x00011ff8`.

Given a shared library and its pseudo-library, the quasi-static linking mechanism works as follows. Pseudo-libraries are statically linked to an application, which is possible since they have the same format as a static library. As a result, pseudo-

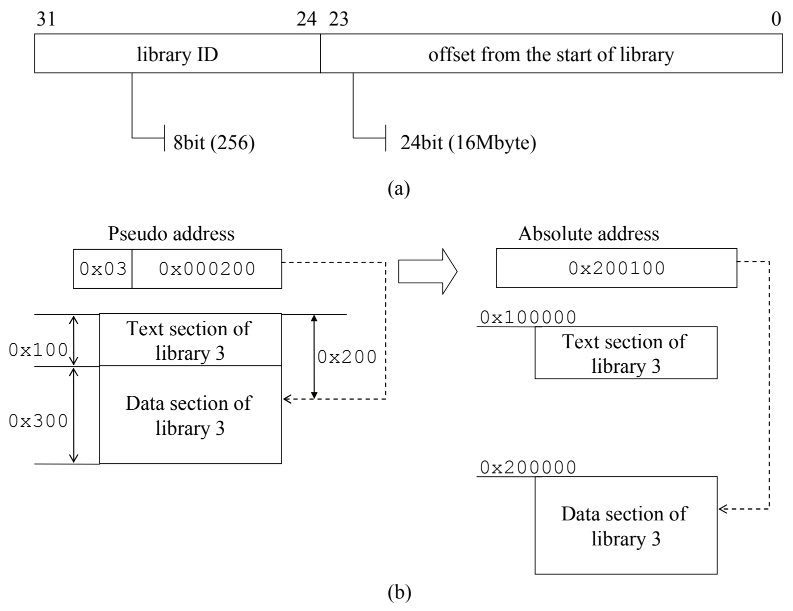

31
24 23
0
library ID
offset from the start of library
8bit (256)
24bit (16Mbyte)
(a)
Pseudo address
0x03
0x000200
Text section of library 3
Data section of library 3
0x100
0x300
0x200
Absolute address
0x200100
0x100000
Text section of library 3
0x200000
Data section of library 3
(b)

Fig. 4. (a) Format of a pseudo-address and (b) an example of converting pseudo address `0x03000200` into an absolute address `0x200100`.

addresses from pseudo-libraries are embedded in locations in an application where the absolute addresses of global symbols are expected. Also, for those locations, relocation entries are created accordingly in the relocation table of the application: a relocation entry is merely an offset inside the application. It is important to mention that there is no need to maintain a symbol table in our shared library because symbols are already bound to pseudo-addresses at linking time. At loading-time, the embedded pseudo-addresses are converted to absolute addresses by the loader after the shared libraries have been loaded into memory.

Specifically, for each relocation entry in the relocation table, the loader first extracts the library ID field from the pseudo-address and then loads the corresponding shared library if it is not loaded yet. Next, it extracts the offset field from the pseudo-address and adds the offset to the base address of the text section of the shared library that were just loaded. Note that the distance in memory between the text section and the data section should be added to this result if the offset is

```
SYMBOL TABLE:
   00000000 l    d  ._shared_lib_symbols_  00000000
   00000000 l    d  .text  00000000
   00000000 l    d  .data  00000000
   00000000 l    d  .bss   00000000
   00000000 l    d  .comment       00000000
   00000000 l    d  *ABS*  00000000
   00000000 l    d  *ABS*  00000000
   00000000 l    d  *ABS*  00000000
   00000000 l    df *ABS*  00000000 _gen_sym_base_.c
   01011ff8 g       ._shared_lib_symbols_  00000000 printf
```

Fig. 3. Symbol table for `printf.o` in the pseudo-library of `uClibc` (excerpted from the result of the command `'arm-uclinux-objdump -t libc.a'`).

larger than the size of the text section, in other words, if the offset points to a memory address in the data section. Finally, the loader replaces the pseudo-address with the converted absolute address. Fig. 3 (b) shows an example of deriving the absolute address from a pseudo-address.

The quasi-static linking mechanism exploiting pseudo-addresses makes the dynamic linking process simpler and faster than true dynamic linking that uses a symbol table. However, quasi-static linking carries the following restrictions, which we believe are not critical limitations for a small embedded system.

- The maximum number of libraries that can be in a system is 255. (The library ID field of a pseudo-address is 8 bits long and ID `0` is reserved for the application program.)
- The library size including text and data sections cannot exceed 16 MB. (The offset field of a pseudo-address is 24 bits long.)
- If a library is updated, all application programs using the library must be re-linked with it. (The offsets of symbols inside the library will change.)

3.2 Execution-In-Place (XIP)

Similar to the quasi-static library technique, we have kept the following three design requirements in mind in designing our XIP technique for MMU-less embedded systems.

CX1. An application program should be linked without requiring prior knowledge of its location in flash memory. (Otherwise, an additional linking would have to be performed after the location is determined. This would increase the complexity in building a file system image.)

CX2. The text section of a program should not be modified even at loading time. (Writing a modified program into flash memory consumes time and energy. Furthermore, file systems are often read-only.)

CX3. The data section of a program should be loaded at an arbitrary location in RAM. (Otherwise, it would be impossible to provide separate data sections for each process without support from an MMU. This would prohibit a program from being executed by multiple processes.)

While the conventional PIC technique easily satisfies requirements CX1 and CX2, it cannot fulfill requirement CX3 for MMU-less systems. PIC generally requires a data section to be located at a fixed distance from a text section since the data section is accessed in a PC-relative manner. In our enhanced PIC technique, the data section can be placed at an arbitrary distance from the text section since the data section is accessed indirectly via a DSBR.

In addition to these requirements, it is still necessary to modify the CRAMFS implementation of uClinux to support XIP. In uClinux, the kernel loader relies on the `mmap()` function of a file system to execute a program in place. Unfortunately, the current implementation of the CRAMFS does not provide the `mmap()` function and thus the kernel loader always copies the text section of a program into RAM and the program is executed from the copied image. As a result, multiple copies of a text section are created and exist in RAM when a program is executed by multiple processes. This presents a significant waste in memory.

Thus, we have implemented our own `mmap()` function into the CRAMFS as follows: (1) For an uncompressed file, our `mmap()` returns the absolute address (i.e., flash memory address) of the file's contents. It is the start address of its text section. (2) For a compressed file, our `mmap()` realizes a functionality that maintains only one copy of the file in RAM.

Table II. Modified components and added functionalities.

`gcc`	- Accepts library ID value via command argument. - Adds instructions in prologue and epilogue of global functions. Those instructions are for saving, setting and restoring DSBR with the help of DSBT.
`gensym`	- Creates pseudo-library which contains only symbol names and pseudo-addresses.
loader	- Creates and initializes DSBT. - Converts pseudo-address to absolute address. - Loads libraries on demand.
CRAMFS file system	- Performs file-to-memory mapping.

Actual decompression and copy from flash memory into RAM occur only once for the first `mmap()` call and subsequent `mmap()` calls just return the address of the copy. These modifications are checked into the source tree of CRAMFS and released together.

Since our XIP technique is orthogonal to quasi-static linking, it can be used in combination with either quasi-static linking or conventional static linking. These two options need a trade-off analysis. Whereas the former yields smaller flash memory usage than the latter since library code cannot be shared under static linking, it leads to larger RAM usage than the latter since data sections of entire library routines are allocated to a process under quasi-static linking. In our implementation, developers can choose both options selectively.

4. Putting It All Together

The proposed mechanisms affect the several stages of the program and library build process including compilation, linking, creation of file system images (i.e., deploying applications and libraries to the file system) and loading. Thus, applying these mechanisms to embedded system development involves modifying the linker, the `elf2flt` transformer and the file system as well as the `gcc` compiler and loader. Table II summarizes the modified tool components and their added functionalities. When we develop a tool chain by integrating the tool components, we emphasize maintaining compatibility with conventional static libraries so that existing applications need not be rewritten. We explain how these tools are implemented and how compatibility is maintained. We also demonstrate how applications and libraries are built via the integrated tool chain, deployed to the file system and loaded at runtime.

Fig. 5 shows our new process for building shared libraries and applications, as well as the tool components that participate in each step. As shown in the figure, most of the build process is performed with object files in ELF (Executable and Linking Format) [16]. This is because the target system's file format (Binary FLaT file format, BFLT [17]) cannot provide information needed for linking. After object files are linked together to create libraries and applications, the resultant executable is converted to BFLT by the `elf2flt` tool. In this section, we explain in detail the process of building and loading shared libraries and applications.

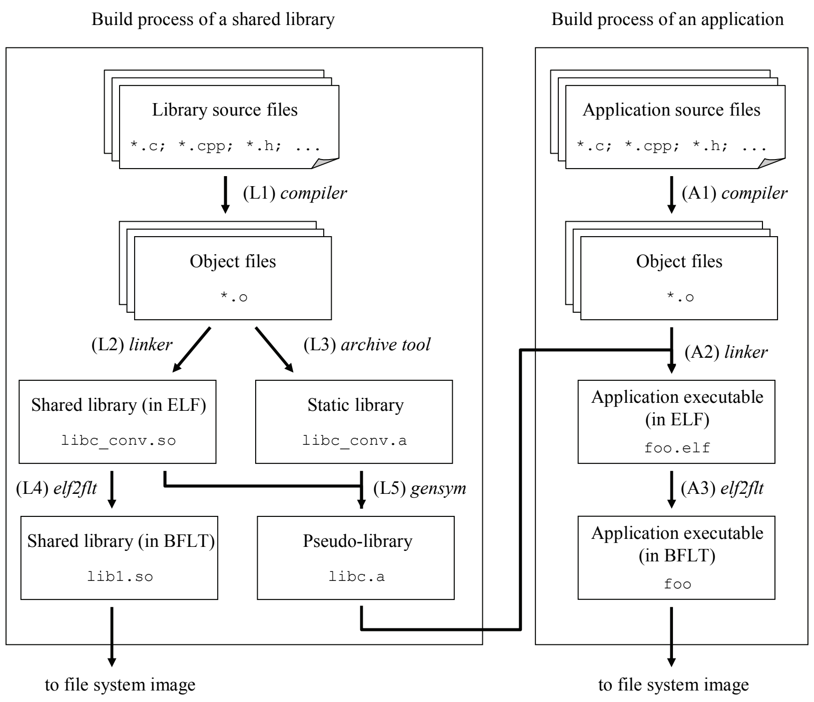


Fig. 5. Build process of a shared library and an application when using proposed mechanisms.

- **Shared library build process**

As shown in Fig. 5, this process accepts one or more library source files as input and produces both a shared library and a pseudo-library as output. In doing so, a normal static library is temporarily generated since it is required by the pseudo-library generation tool as input. The static library and the shared library are built following exactly the same process as in a conventional library system. First, our modified `gcc` compiler compiles library source files and generates object files in PIC. This compilation step is controlled by a command line option (`-mquasi-static`) which has been added to the original compiler. With this option, as explained in Section 3.1.2, code that fetches a data section base address to a DSBR is generated at the prologue of each global function and code that restores the DSBR is generated at the epilogue. The programmer-supplied library ID is passed through a command line argument (`-mshared-library-id <id>`) that we have extended into the `gcc` compiler. Second, the linker links all object files together to generate a shared library in ELF (denoted as L2 in Fig. 5). The shared library is, then, transformed into BFLT so that it is loadable into the target system. Third, the archive utility (normally `ar`) collects object files and produces a static library in ELF (denoted as L3 in Fig. 5) which, as explained earlier, is temporarily generated to create a pseudo-library.

The final step of the library build process creates a pseudo-library (denoted as L5 in Fig. 5). We have developed a new tool named `gensym` for this step. As depicted in Fig. 6, `gensym` takes a static library and a shared library both in ELF as input. The pseudo-library is created by extracting symbol tables from this static library and discarding the rest. All symbols in the pseudo-library are then redefined with pseudo-addresses generated from the library ID and the offsets provided in the shared library, as explained in Section 3.1.3. Pseudo-libraries built throughout this process are placed in the development

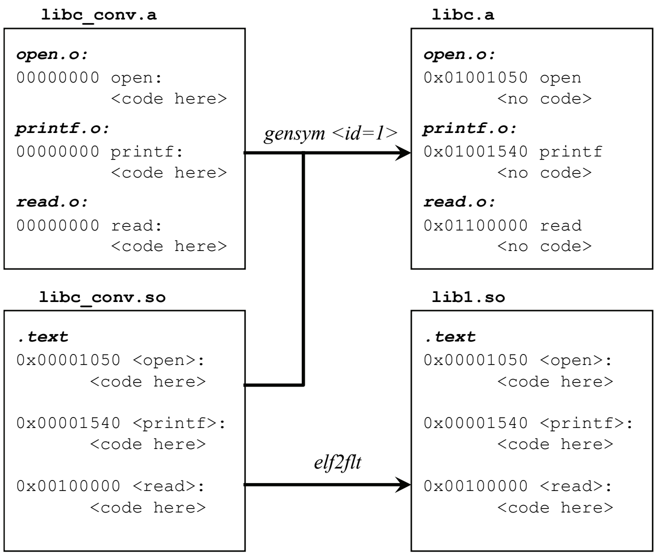


Fig. 6. Creation of pseudo-library.

environment so that they can be quasi-statically linked with applications. Since they have the same format and names as conventional static libraries, the existence of pseudo-libraries is transparent to application developers.

- **Application build process**

Compilation of non-XIP applications is not affected by our techniques except that a register designated as a DSBR is not used as a general-purpose register. For XIP applications, our compiler runs with additional options (`–mquasi-static` and `-mshared-library-id <id>` with `0` as its id) that are identical to those used for compiling a quasi-static shared library. Besides, when linking an application with a pseudo-library, our linker (denoted as A2 in Fig. 5) always runs with option "`–mquasi-static`" regardless of whether or not the application is for XIP.

- **File system creation**

Application programs and shared libraries are stored in the CRAMFS file system. If a program or a shared library is to be executed in place, an associated file needs to be uncompressed so that it can be directly mapped to memory. As previously mentioned, file compression is controlled by a sticky bit of a file. A tool creating a CRAMFS file system image does not compress a file whose sticky bit is on.

- **Loading process**

The loading process involves the following four steps: (1) loading the executable file of an application program, (2) performing relocations, (3) creating DSBTs and (4) initializing the DSBR. Specifically, the kernel loads an executable file by calling the `mmap()` function for the text section and creating the in-memory copies of the data and bss sections. As explained earlier, our implementation of the `mmap()` function does not copy the text section if the executable file is designated for XIP. After loading an executable file, the kernel performs relocations by converting each pseudo-address into an absolute address. In doing so, the kernel extracts library ID from a pseudo-address and loads a corresponding shared library, if it has not yet been loaded. In our system, searching for a shared library in a file system is not necessary since a shared library has a pre-assigned name, `lib<id>.so`, and always resides in the "`/lib`" directory in a file system.

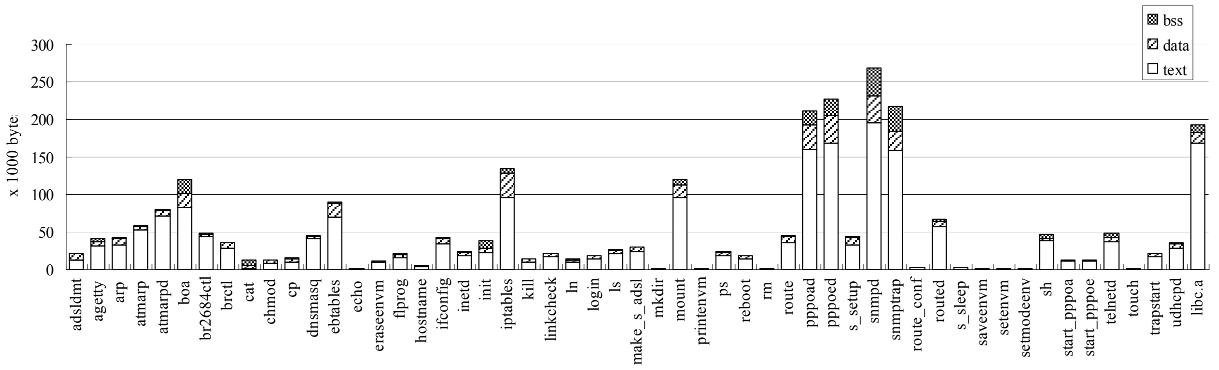


Fig. 7. Home network gateway applications decomposed into their text, data, and bss sections.

After loading a shared library, the kernel relocates symbols in the library. The symbol relocation is performed recursively when there are external references to the symbols in other library. In this case, the external symbols must also be relocated. The recursion continues until no external reference is found.

After the relocation, the kernel creates multiple copies of the DSBT and places each of them just before the data section of each shared library. In addition to this, if the application program is compiled using the enhanced PIC technique, another DSBT is created and placed before the data section of the program itself. Note that each entry in the DSBT contains the data section base address of a corresponding library. Subsequently, the DSBR is initialized as follows. For an application program compiled using the enhanced PIC technique, the DSBR is initially set to the data section base address of the program. In contrast to this, for other applications, the data section base address of the first shared library is used for the initial value of the DSBR.

5. Experimental Evaluation

We have implemented the proposed shared library and XIP techniques in a commercial ADSL home network gateway equipped with an MMU-less ARM7TDMI processor core, 2-MB NOR flash memory and 16-MB SDRAM. We have conducted a series of experiments to measure its memory usage and evaluate its performance overhead. Before presenting the measured results, we will give an overview of the target application programs and libraries.

The target system contains three classes of application programs: system utilities such as `cp` and `mount`, modem utilities such as `adsldmt` and communication utilities such as the `boa` web server. Fig. 7 shows how each application program is decomposed into its text, data and bss sections. These application programs use two libraries: the standard C library `uClibc` (`libc.a` in Fig. 7) that is distributed with uClinux and the SNMP library `UCD-SNMP`. We applied the shared library technique only to `uClibc` since `UCD-SNMP` is not shared among application programs but is used by only two SNMP-related utilities: `snmpd` and `snmptrap`.

5.1 Measuring Memory Usage

Since our overall aim is to reduce power consumption and production costs of the system by reducing its memory requirement, the metrics of our experiments have focused on flash memory and RAM usage. Flash memory usage is measured as the aggregated size of each application program and shared library files in the CRAMFS. Similarly, RAM

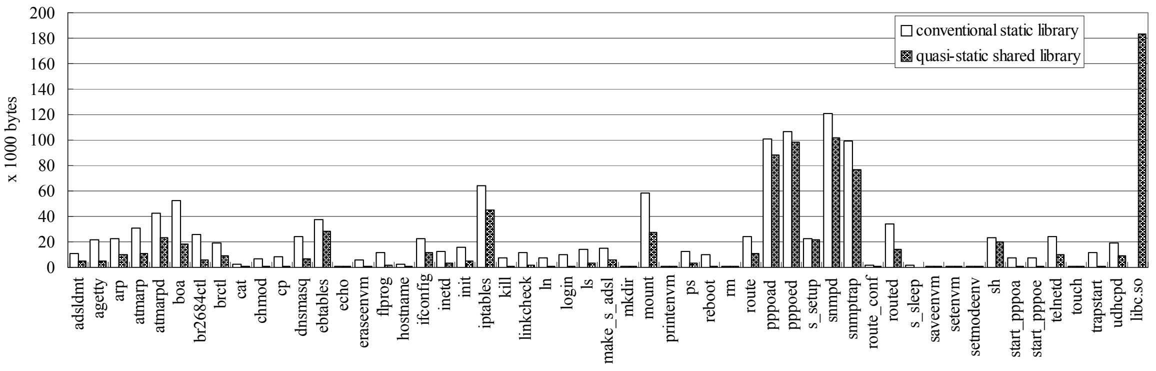


Fig. 8. Flash memory usage of each application and shared library.

usage is defined as the size of RAM that application programs and shared libraries occupy. In order to show the net effect of using our techniques, we intentionally exclude heap and stack sizes from RAM usage since they are not affected by our techniques. When we report on RAM usage reductions in percentages, we provide two numbers: one including heap and stack sizes and another excluding them. Our experiments were conducted for two cases, one where shared library technique is applied alone and the second where both the shared library and XIP techniques are applied together. In this section, we report on the measured results for each case.

5.1.1 Case 1: Quasi-Static Shared Library Only

Fig. 8 depicts the flash memory usage of each application program and the shared library file. Note that the shared library file `libc.so` has only the gray bar since it does not exist in the final file system image in the conventional static library technique. Our shared library technique reduces overall flash memory usage by 35% (from 1173 KB to 768 KB) by reducing the flash memory usage of each application program. The reason behind this is clear: library code is not copied into the executable file of each application program but is instead kept as a single shared library file.

Fig. 9 depicts the RAM usage of each application program and the shared library. Compared to the decrease in flash memory usage, the reduction in overall RAM usage is marginal: it is decreased by 3% from 2867 KB to 2794 KB. If we include the space for the heap and stack, which measures 1114 KB, the reduction is 1.9%. Such a small amount of

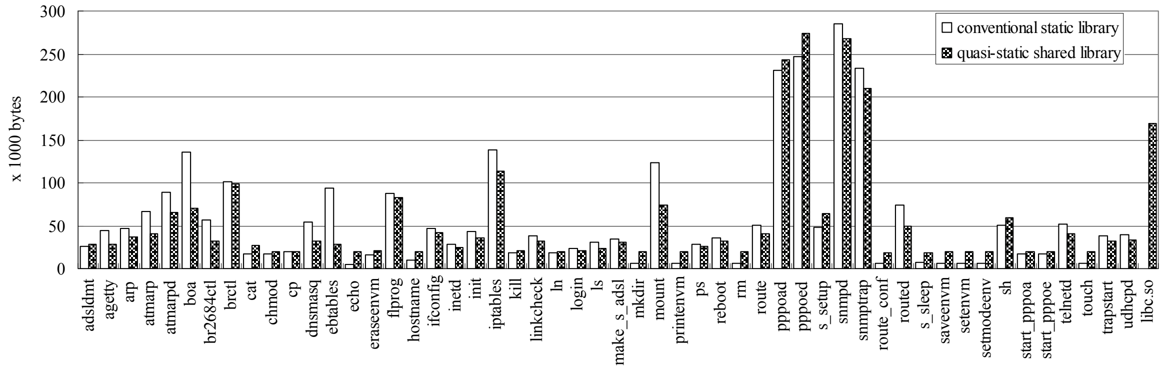


Fig. 9. RAM usage of each application and shared library.

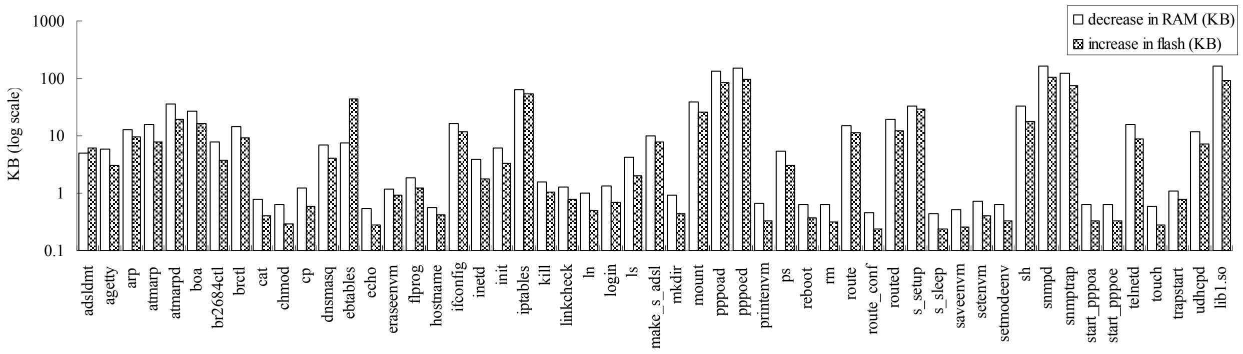


Fig. 10. Decreases in RAM usage and increases in flash memory usage of each application and library when XIP is applied.

reduction is caused by a hidden waste in RAM usage that the use of a shared library technique incurs. The waste occurs since the data sections of entire library routines are allocated to a process in shared library technique, while only the data sections of specific library routines, which are linked to an application program, are allocated to a process in the conventional static library technique. For this reason, if the data section of an entire library is larger than the size of the actually used library code, the use of shared library technique yields no advantage in terms of RAM usage. For our application programs and shared libraries, the reduction of RAM usage is marginal since the hidden waste is almost as big as the reduction in the RAM usage.

However, this does not mean that our quasi-static shared library technique cannot effectively reduce the overall RAM usage. In our implementation, it is possible for developers to selectively choose for each application program whether it is linked to quasi-static shared libraries or to conventional static libraries. To minimize RAM usage, quasi-static shared libraries can be applied only to those application programs that incur only a small hidden waste.

5.1.2 Case 2: Quasi-Static Shared Library and XIP Together

In order to reduce RAM usage, developers can take advantage of XIP though flash memory usage will increase. Note that an application program or a library must not be compressed in a file system in order for it to be executed in place. Fig. 10 shows the decrease in RAM usage and the increase in the flash memory usage of each application program, and the library, when XIP is applied. When we execute all of them in place, overall RAM usage is decreased by 42% from 2794 KB to 1631 KB, while flash memory usage is increased by 102% from 768 KB to 1555 KB. The percentage of RAM usage reduction is 30% when the stack and heap sizes are included.

When the size of flash memory is limited, it may not be desirable to apply the XIP technique to all application programs and libraries in the system. In our case, only 634 KB out of the 2-MB flash memory are available after the quasi-static shared library technique is applied: 128 KB and 518 KB are already used by the `ARMboot` boot-loader and the uClinux kernel, respectively and the file system takes up 768 KB. Therefore, we need to select application programs and libraries for XIP with a flash memory budget of 634 KB. This is a binary knapsack problem that is NP-hard.

Thus, we have developed a heuristic algorithm to make our selection. The selection is based on following two criteria. First, libraries and programs that are always executed are preferred; if they are not selected for XIP, a fixed amount of

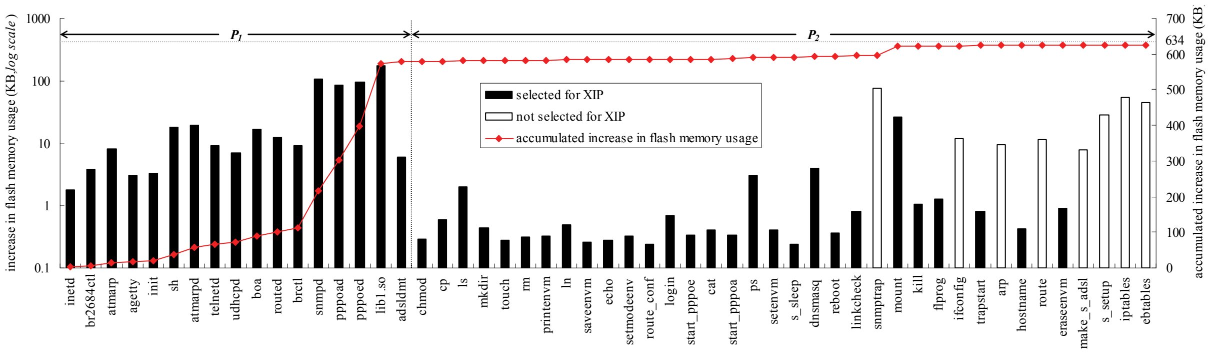


Fig. 11. The accumulated increase in flash memory usage with our heuristic algorithm.

RAM is constantly consumed by them. Second, programs and libraries that can save more RAM with less flash memory get higher precedence since our goal is to maximize RAM reduction with a given flash memory budget. Our algorithm works in three steps. First, libraries and programs are categorized into two groups: those that are always executed (P_1), and those that are not (P_2). Second, programs and libraries are ordered as follows. Those that are in P_1 always come first and those that are in P_2 come later. In addition, within a group, each element is sorted in decreasing order of the ratio of benefit (decrease in RAM usage) to unit cost (flash memory usage). Finally, the algorithm repeatedly chooses a program or a library for XIP in the sorted order until no more can be included. If one of the programs or libraries cannot fit into available flash memory, the algorithm skips it and proceeds with the next program or library in the sorted order.

Fig. 11 depicts a set of application programs and a library sorted in accordance with our heuristic algorithm; their individual increase in flash memory usage when XIP is applied (depicted as bars); and the accumulated increase in flash memory usage (depicted as a line with dots) when our heuristic algorithm is applied. Using our algorithm with a flash memory budget of 634 KB, eight programs (programs with a blank bar in the figure) are not selected for XIP.

If we assume that all programs and libraries are simultaneously running, resultant RAM usage is decreased by 32% from 2794 KB to 1911 KB. It is a reduction of 23% if we include the stack and heap sizes in the RAM usage. On the other hand, when we measured the worst-case RAM usage at run-time, the RAM usage reduction was actually 30% from 2748 KB to 1932 KB, which includes the heap and stack sizes.

5.2 Measuring Performance Overhead

The ADSL home network gateway has two operational scenarios: one is the normal operational scenario in which the target system performs protocol conversions from ATM traffic to Ethernet traffic and vice versa; the other is the configuration scenario in which the user changes the configuration and monitors the status of the target system. In the normal operational scenario, applying our shared library and XIP techniques does not affect the performance of the target system because all the protocol conversions are performed by the kernel, not by user programs. We therefore turn our attention to the configuration scenario.

In order to evaluate the performance penalty caused by our techniques, we measured the response time for configuring the target system. In our system, a user configures the gateway using a web browser. The request is serviced by the

Table III. Response time of configuring the target system via a web browser.

	with conventional static library	with quasi-static shared library and XIP
response time	957 ms	994 ms

`adsl.cgi` program via a `boa` web server. In order to apply the changes in the configuration, the `adsl.cgi` program invokes other system utilities such as `sh`, `ifconfig` and `iptables` several times (eight times for `sh`, seven times for `ifconfig`, and once for `iptables`). Table III shows the result with the response time representing the time taken for the execution of `adsl.cgi`. The result shows that the performance penalty for the shared library and XIP is only 3.4%.

The reduction of loading time in these programs greatly contributes to this result. Since they were compiled using the enhanced PIC technique, reference relocation for local static data and functions are totally eliminated. Fig. 12 shows that the amount of time taken for loading a program is decomposed into two: time taken for loading the program itself and time taken for loading the libraries. We can infer from the figure that the total loading time for most application programs is decreased by an average of 59%.

6. Related Work

Research into memory footprint reduction in embedded systems has been mainly focused on four techniques: demand paging, compression, shared libraries and XIP techniques. Demand paging is slowly adopted in embedded systems with the advent of the page replacement algorithms that can effectively reduce the performance degradation that could otherwise be a critical limitation of demand paging in embedded systems. These algorithms include Clock-Pro [18], DEAR [19], ARC [20], CAR [21], LRFU [22] and EELRU [23]. However, their utility is still limited in embedded systems since they rely on paged virtual memory, which requires an MMU and often hard disks.

This limitation has led several researchers to devise demand paging techniques that are applicable to systems without an MMU. Specifically, Park et al. in [24] propose a compiler-assisted demand paging technique. In this technique, a post-pass compiler analyzes a program binary and transforms function call and return instructions into calls to a page manager. Upon

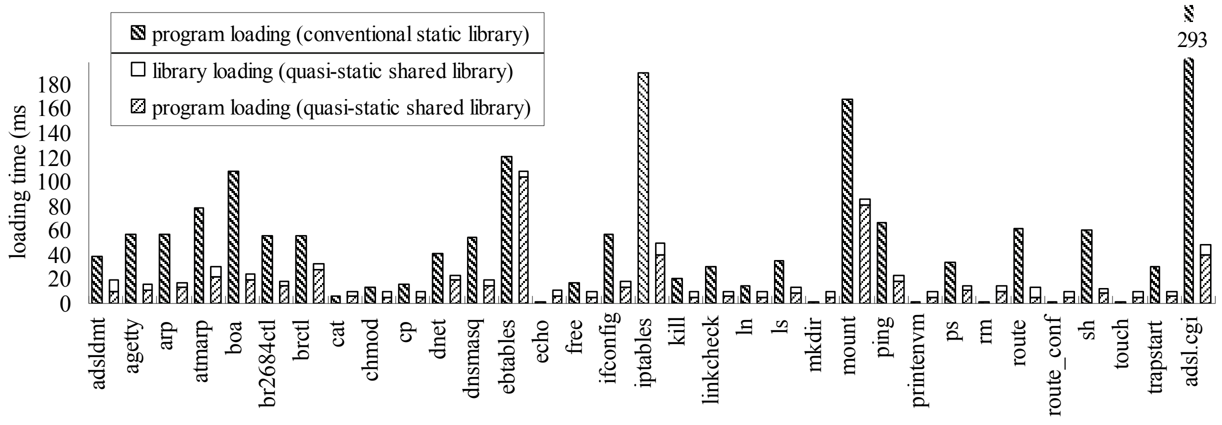


Fig. 12. Loading time decomposed into shared libraries loading and program loading.

being invoked, the page manager loads the code of each function into memory. With this technique, they succeed in reducing the code size in RAM by 33%. However, this technique cannot scale up with larger memory and multiple tasks because a software-managed page table must be maintained and address translation using this table incurs a significant performance overhead. In [25], a similar technique is proposed based on an application-level virtual memory library and a virtual memory aware assembler. However, it also has all of the aforementioned problems of the compiler-assisted demand paging.

Compression-based memory footprint reduction techniques can be categorized into hardware-based and software-based compression techniques. In hardware-based techniques, instructions and/or data are compressed and decompressed at run-time by dedicated hardware units [26-30]. Since the hardware units run in parallel with the main processor, they incur low performance overhead. However, additional hardware increases the production cost as well as the power consumption and the complexity of a system.

Software-based compression techniques can further be divided into two categories: file system compression and swap compression. CRAMFS (Compressed ROM File System), Cloop (compressed loop-back device) and CBD (Compressed Block Device) are examples of file system compression techniques. Compressed file systems are particularly useful for saving space that a file system occupies in storage. In addition, CRAMFS provides an essential feature essential XIP that stores selected files in uncompressed form. For this reason, our proposed technique is also based on CRAMFS as mentioned earlier.

Swap compression techniques presented in [31-34] increase effective memory capacity by compressing infrequently used pages and storing them in a swap area in RAM. However, they still rely on an MMU for decompressing a swapped-out page when the page is accessed again. This limitation led Bai at el. [35] to propose a technique called MEMMU. In this technique, every data object is restricted to be accessed indirectly via its handle. Since an access function of this handle loads and unloads the data object on demand, an MMU is not required. However, runtime check and management of mappings from handles to memory locations are required, incurring substantial performance overhead.

Shared libraries have played an integral role in reducing the memory footprint of a system since its introduction in Multics [36] and TENEX [37]. Traditionally, shared libraries have been simply classified into statically linked and dynamically linked shared libraries [6, 8, 36-38]. Even though the former incur less run-time overhead, the latter have become the dominant practice due to their flexible memory use and easy library updates. In fact, they are used by most popular operating systems including Windows [39-41], Linux [41, 42], Unix [9] and BSD derivatives. On the other hand, unlike the others, Windows operating systems try to alleviate the performance overhead of dynamically linked shared library techniques by statically allocating a fixed address to each of the DLLs (dynamically linked libraries) [39]. As such, a DLL is dynamically linked only when the allocated address of the DLL conflicts with those of other DLLs.

Without significant redesign, none of the above shared library approaches can be applied to MMU-less embedded systems since they were originally designed to use features provided by an MMU. A dynamically linked shared library framework proposed by Cadenux [12] is one of only a few shared library techniques designed for MMU-less systems. This technique is similar to ours in the sense that an MMU is emulated using a dedicated register identical to the DSBR. However, compared to our technique, it has several limitations. First, a stub code that manages the value of the register is generated for each library function and included in the caller, instead of the callee. Thus, a significant amount of memory is wasted for duplicated stub code especially when multiple programs use the same library function. Second, it does not support GOT, meaning that global variables can not be used directly, but indirectly through access functions. This requires a significant amount of changes be made in existing application programs. Third, a shared library has to include symbol tables that take up valuable space both in a file system and memory. Finally, time-consuming dynamic linking using the symbol tables must be performed at loading time and this is done every time a program runs. Thus, a program has to deal with substantial performance cost. Those limitations clearly make the technique inappropriate for resource-constrained embedded systems in practice.

XIP [13, 14] is a technique whereby a program stored in ROM or flash memory is run directly from the location where it is stored. In MMU-less systems, XIP has been used only for operating system kernels whose data section is assigned a statically determined fixed address in RAM. Note that assigning a fixed address to each data section of XIP programs creates a significant waste in memory usage. Furthermore, it is still impossible to execute the same program by multiple processes simultaneously. As explained, we address this problem by using the enhanced PIC technique used in our quasi-static shared library. To the best of our knowledge, there has been no XIP technique that supports executing multiple instances of an application program in MMU-less systems.

7. Conclusions

In this paper, we have presented the quasi-static shared library and XIP techniques as a means of reducing both the dynamic and static memory requirements of heavily optimized, resource-constrained embedded systems lacking an MMU. The proposed techniques are based on our quasi-static linking mechanism in which global symbols referenced in a library are bound to pseudo-addresses at linking time and the actual physical addresses are bound at loading time. Quasi-static linking is distinct from existing shared library techniques that rely on either normal static linking or dynamic linking because it utilizes static linking only partially for pseudo-addresses and does not employ a symbol table, which dynamic linking does. Avoiding symbol tables allows our technique to incur only a small amount of time and space overhead. This is possible because pseudo-addresses stored in pseudo-libraries keep the information required for symbol binding, and absolute addresses can be computed via a very simple address resolution formula involving only simple integer operations. Our technique maintains compatibility with conventional static libraries so that existing applications need not be rewritten. These characteristics make our technique appropriate for COTS-based embedded systems that are subject to stringent performance and resource constraints.

We have implemented our techniques by emulating the memory-mapping feature of an MMU with a DSBR and DSBTs. This required us to modify the `gcc` code generator, the `elf2flt` utility, the uClinux's loader and CRAMFS and to implement a tool called `gensym` for creating pseudo-libraries. We have implemented all of the techniques in a commercial ADSL home network gateway and created a tool chain for building and deploying shared libraries. In doing so, open source software has played a pivotal role. It allows for various source code modifications.

We have performed extensive measurements to analyze the RAM and flash memory requirements of the home network gateway and evaluate its performance overhead. The results are very impressive: 35% reduction in flash memory usage through the shared library technique alone and 32% reduction in RAM usage when using the shared library and XIP techniques together. These results were achieved with negligible performance penalty of less than 4%.

There are two research directions along which our techniques can be extended. First, we are developing a tool that automatically divides a shared library into several smaller shared libraries by inspecting the usage patterns of given applications. We expect that this tool, if used in the quasi-static library technique, will help reduce RAM usage more effectively since unrelated portions of the original shared library do not have to be linked. Because the original shared library is not linked in its entirety, valuable space in RAM is freed up with the only drawback being a slight increase in the number of DSBT entries due to the increased number of shared libraries. Because the DSBT is capable of holding a maximum of 255 entries, this technique should be feasible for the vast majority of embedded systems. The second direction for our future work is to devise an objective policy to automatically select programs for XIP. Although our heuristic algorithm produces acceptable results, we seek to find an optimal solution by formulating the program selection process as an optimization problem. The results look promising.

References

[1] J. Lee, J. Park, and S. Hong, "Memory Footprint Reduction with Quasi-Static Shared Libraries in MMU-less Embedded Systems," in The 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), San Jose, California, USA, 2006.

[2] Arcturus Networks Inc., "uClinux: Embedded Linux and Microcontroller Project," URL: http://www.uClinux.org.

[3] D. McCullough, "uCLinux for Linux programmers," *Linux Journal*, vol. 2004, 2004.

[4] N. Wells, "BusyBox: A Swiss Army Knife for Linux," *Linux Journal*, vol. 2000, 2000.

[5] J. A. C. Bingham, *ADSL, VDSL, and Multicarrier Modulation*: John Wiley & Sons, Inc., 2001.

[6] J. R. Levine, *Linkers and Loaders*: Morgan Kaufmann, 2000.

[7] M. Sabatella, "Issues in Shared Library Design," in Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, USA, 1990.

[8] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared Libraries in SunOS," in Proceedings of the USENIX 1987 Summer Conference, Phoenix, Arizona, USA, 1987.

[9] J. Q. Arnold, "Shared Libraries on UNIX System V," in Proceedings of the USENIX 1986 Summer Conference, 1986.

[10] D. M. Beazley, B. D. Ward, and I. R. Cooke, "The Inside Story on Shared Libraries and Dynamic Loading," *IEEE Computing in Science and Engineering*, vol. 3, pp. 90-97, 2001.

[11] M. L. Scott, *Programming Language Pragmatics*, 2nd ed. San Francisco: Morgan Kaufmann, 2006.

[12] RidgeRun Inc., "Cadenux XFLAT Shared Libraries," URL: http://www.ridgerun.com/XflatPages/CadenuxXFLATSharedLibraries.html, 2006.

[13] D. Verneer, "eXecute-In-Place," in *Memory Card Magazine*, 1991.

[14] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, and B. Kim, "A Low-Cost Memory Architecture with NAND XIP for Mobile Embedded Systems," in First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003.

[15] R. M. Stallman, Using and Porting the GNU Compiler Collection (GCC) for GCC Version 2.95: Free Software Foundation, 1999.

[16] Tools Interface Standards (TIS), "Executable and Linkable Format (ELF) Specification, version 1.2," Portable Formats Specifications 1995.

[17] C. Peacock, "uClinux: BFLT Binary Flat Format," URL: http://www.beyondlogic.org/uClinux/bflt.htm, 2005.

[18] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement " in USENIX 2005 Annual Technical Conference, Anaheim, CA, 2005.

[19] J. Choi, S. H. Noh, S. L. Min, E.-Y. Ha, and Y. Cho, "Design, Implementation, and Performance Evaluation of a Detection-Based Adaptive Block Replacement Scheme," *IEEE Transactions on Computers*, vol. 51, No. 7, 2002.

[20] N. Megiddo and D. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache," in Proceedings of the 2nd USENIX Symposium on File and Storage Technologies, 2003.

[21] S. Bansal and D. Modha, "CAR: Clock with Adaptive Replacement," in Proceedings of the 3rd USENIX Symposium on File and Storage Technologies, 2004.

[22] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies " in Proceedings of 1999 ACM SIGMETRICS Conference, 1999.

[23] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," in Proceedings of the 1999 ACM SIGMETRICS Conference, 1999.

[24] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min, "Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory " in Proceedings of the 4th ACM international conference on Embedded software EMSOFT '04, 2004.

[25] S. Choudhuri and T. Givargis, "Software Virtual Memory Management for MMU-Less Embedded Systems," Center for Embedded Computer Systems, University of California, Irvine TR 05-16, Nov. 2005.

[26] H. Lekatsas, J. Henkel, and W. Wolf, "Code Compression for Low Power Embedded System Design " in Proceedings of Design Automation Conference, 2000.

[27] L. Benini, D. Bruni, A. Macii, and E. Macii, "Hardware-Assisted Data Compression for Energy Minimization in Systems with Embedded Processors," in Proceedings of the Conference on Design, Automation and Test in Europe 2002.

[28] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT) " *IBM Journal of Research and Development*, 2001.

[29] Advanced RISC Machines Ltd., "An Introduction to THUMB," March 1995.

[30] X. H. Xu, C. T. Clarke, and S. R. Jones, "High Performance Code Compression Architecture for the Embedded ARM/THUMB Processor," in Proceedings of the 1st Conference on Computing Frontiers Ischia, Italy 2004.

[31] S. Roy, R. Kumar, and M. Prvulovic, "Improving System Performance with Compressed Memory," in Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01) 2001.

[32] T. Cortes, Y. Becerra, and R. Cervera, "Swap Compression: Resurrecting Old Ideas," *Software-Practice and Experience*, vol. 30, pp. 567-587, 2000.

[33] L. Rizzo, "A Very Fast Algorithm for RAM Compression," *ACM SIGOPS Operating Systems Review* vol. 31, pp. 36-45, 1997.

[34] L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar, "On-Line Memory Compression for Embedded systems," *to appear in ACM Trans. Embedded Computing Systems*, 2007.

[35] L. Bai, L. Yang, and R. P. Dick, "Automated Compile-Time and Run-Time Techniques to Increase Usable Memory in MMU-Less Embedded Systems," in Proc. Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems, 2006.

[36] E. I. Organick, The Multics System: An Examination of Its Structure: MIT Press, 1972.

[37] D. L. Murphy, "Storage Organization and Management in TENEX," in Proceedings of the Fall Joint Computer Conference, AFIPS., 1972.

[38] AT&T, *System V Application Binary Interface*. Upper Saddle River, NJ: UNIX Press/Prentice Hall, 1990.

[39] M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format," in *MSDN Magazine*, vol. 17, 2002.

[40] Microsoft co., "Visual Studio, Microsoft Portable Executable and Common Object File Format Specification, Revision 8.0," URL: http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, May 2006.

[41] R. Thomas and B. Reddy, "Dynamic Linking in Linux and Windows," URL: http://www.securityfocus.com/infocus/1872, 2006.

[42] D. M. Beazley, B. D. Ward, and I. R. Cooke, "The Inside Story on Shared Libraries and Dynamic Loading," *Computing in Science & Engineering* vol. 3, pp. 90-97, 2001.

[43] J. Goodman, W. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," in Proceedings of the 2nd International Conference on Supercomputing, pp. 442-452, 1988.

[44] S. Mahlke, W. Chen, P. Chang, and W. Hwu, "Scalar Program Performance on Multiple-Instruction-Issue Processors with a Limited Number of Registers," in Proceedings of the 25th Annual Hawaii International Conference on System Sciences, 1992.

[45] ARM Limited, Procedure Call Standard for the ARM Architecture, 2007.