Memory Footprint Reduction with Quasi-Static Shared Libraries in MMU-less Embedded Systems

Jaesoo Lee, Jiyong Park, and Seongsoo Hong
School of Electrical Engineering and Computer Science,
Seoul National University, Seoul 151-742, South Korea
{jslee, parkjy, 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 is known to be effective in significantly reducing memory usage. Unfortunately, many resource-constrained embedded systems lack MMU, making it extremely difficult to support this technique. To address this problem, we propose a novel shared library scheme called the quasi-static shared library. In quasi-static shared libraries, global symbols are bound to pseudo-addresses at linking time and the actual physical addresses are bound at loading time. This scheme is made possible by emulating MMU's memory mapping feature with a Data Section Base Register (DSBR) and a Data Section Base Table (DSBT). Quasi-static shared libraries do not require symbol tables which take up time and space at runtime.

We have implemented the proposed scheme in a commercial ADSL (Asymmetric Digital Subscriber Line) home network gateway and conducted a series of experiments measuring its memory usage and performance overhead. The result is drastic: a 35% reduction in flash memory usage and a 10% reduction in RAM usage. These results were achieved with only a negligible performance penalty of less than 4%. Even though this scheme was applied to uClinux-based embedded systems, it can be used for any MMU-less real-time operating system.

1 Introduction

Embedded systems are generally characterized as systems which have stringent constraints on resources such as processing power, power consumption, 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 as a combination of COTS (Commercial Off-The-Shelf) software components such as Linux-based operating systems [1] and the BusyBox multi-call utility [2]. It is technically very hard to fit heavily pre-optimized COTS software components into systems with tight memory constraints. Furthermore, this requires sometimes unexpected and often error-prone tasks like modifying the compiler, linker, loader, and operating system. In this paper, we present an effective technique, the quasi-static shared library, to reduce memory usage in MMU-less embedded systems. Also, we present the results of applying it to a commercial ADSL (Asymmetric Digital Subscriber Line) [13] home network gateway.

A shared library [3][8][9][10][11], in general terms, is a library that contains routines that are loaded into memory by the operating system as needed and dynamically shared with other applications without static linking. If a shared library is used, an application need not contain the routines in that library. Instead, it can share with other applications a copy of those routines which is already loaded into memory. This helps effectively reduce the amount of file system space and runtime memory used and allows shared library routines to be replaced on-the-fly. MMU is usually required to implement a shared library. A library routine is composed of a public code section that is shared among processes and a private data section that is assigned separately to each process. When a library routine accesses a data symbol, it must be redirected to the data's physical address that is specific to each process. To achieve this, the data section of a library should be mapped to a fixed area in the address space of each process. This mapping can be done easily with MMU. However, MMU is rarely used in resource-constrained embedded systems since it greatly increases both the production cost and the complexity

of a system. Thus, shared libraries are not commonly implemented in embedded systems as compared to workstations or desktop PCs that contain MMU.

Before we consider the implementation of a shared library on an MMU-less embedded system, we must first make a distinction between two types of shared libraries based on the point at which global symbols are bound to actual addresses. The first type is the dynamically linked shared library [3] in which symbols are bound to addresses at loading time. The other type is the statically linked shared library [3] in which symbols are bound to addresses at linking time. Shared library frameworks implemented by RidgeRun [5] and Cadenux [5] fall under the category of dynamically linked shared libraries. In this kind of framework, the dynamic loader fixes all the references in an application using symbol tables contained in the shared library. This means that the symbol tables should be maintained throughout the lifetime of the application, taking up valuable space. This also causes a substantial runtime performance penalty due to the time-consuming dynamic linking performed when a process refers to a library symbol for the first time. Hence, dynamically linked shared libraries are undesirable for embedded systems in which memory and CPU resources are restricted. However, statically linked shared libraries are also undesirable because each library must be allocated to a non-overlapping region in a single address space. This is done to avoid conflicts among different libraries but has the potential effect of creating serious memory waste.

To gain the advantages of both dynamically and statically linked shared libraries while avoiding the aforementioned problems, we propose the quasi-static shared library. It 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.

We have implemented the proposed shared library scheme in a commercial ADSL home network gateway. This system is equipped with an MMU-less ARM7TDMI processor core, 2MB flash memory, and 16MB SDRAM. `uClinux` [1], a derivative of Linux for microcontrollers without MMU, is used as the operating system. The target system is very complex, running nearly 11 out of 35 applications simultaneously during the course of its usual operation. We applied our shared library scheme to all 35 of the applications and the one library. We also performed a series of experiments to measure memory usage and evaluate performance overhead. The results show that the proposed schemes reduce memory usage, both in terms of flash memory and RAM, by a considerable amount with negligible performance degradation.

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 quasi-static shared libraries and explains how these mechanisms are coherently integrated into the overall system. Section 4 provides the results of an empirical evaluation of our schemes in the target system. Finally, Section 5 serves as our conclusion.

2 Target System Components and Executable Code Management

To aid in understanding the rest of this paper, we present the target system components and explain the original process of linking and loading executable code and organizing the file system. Table I shows the hardware and software components of the target system. Basically, this system is connected to the Internet through ADSL and provides an internet connection to other home network devices. It also provides information about these devices, drivers for the devices, and a web-based interface to control them.

The original system was constructed with conventional static libraries where library routines are copied and statically linked to applications. When the executable code image of an application is created, symbol addresses are determined with the assumption that executable code will be loaded from address `0`, as in systems with MMU. However, because the target system lacks MMU, it is impossible to provide multiple virtual address spaces starting from address `0` for processes. Thus, the executable code image must have a data structure that enables the modification of symbol addresses that it uses. That data structure is a relocation table and is used by the loader when the executable code image is loaded.

The created executable code images are merged together to create a CRAMFS image which is stored in flash memory. In flash memory, there resides not only the CRAMFS image, but also a bootloader and a compressed image of `uClinux`. The executable code images in CRAMFS may be either compressed or uncompressed according to the developer's choice. In practice, most are compressed.

An executable code image stored in flash memory is loaded into RAM by `uClinux`'s loader. In doing so, the loader calculates the symbols' absolute addresses and modifies the code and data sections using the relocation table.

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

Hardware	Main processor	S5N8947 (ARM7/TDMI core, No MMU, 40MHz)
	ADSL processor	S5N8950 ADSL DSP, S5N8951 AFE
	Memory	16MB SDRAM, 2MB 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 35)
	Libraries	uClibc 0.9.15 (standard C library)
	Development tools	gcc 3.2.1 (compiler), binutils 2.13.1 (linker, assemblers, and etc.), elf2flt (conversion tool from ELF to BFLT), and etc.

3 Design Requirements and Solution Mechanisms

As mentioned in previous sections, it is quite difficult to implement shared libraries in MMU-less embedded systems. Furthermore, we seek to reduce time and space overhead by avoiding dynamic data structures such as symbol tables, 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 [4], and the `uClinux`'s loader. We also have to implement a tool for creating pseudo-libraries that play the role of symbol tables used in conventional shared libraries.

Before getting into these details, we first enumerate the design requirements for quasi-static shared libraries. In designing our schemes for memory-constrained MMU-less embedded systems, we have the following three design requirements in mind.

C1. The libraries should operate regardless of their location, and their text sections should never be modified once they are loaded. (A shared library created with a specific, statically determined load address in RAM might cause a significant waste in memory usage; the memory space assigned to the library cannot be used for other purposes even though the library is not used by any other processes)

C2. Multiple shared libraries should be usable by a single process. (Without this functionality, the shared library would be very restrictive to developers.)

C3. The loader should operate without a symbol table. (We therefore avoid incurring the associated performance penalties and space overhead.)

In Section 3.1, we describe the solution mechanisms to meet these requirements. Subsequently in Section 3.2, we show how these mechanisms are integrated coherently into the overall system.

3.1 Solution Mechanisms

To meet these requirements, a variety of techniques are employed. For condition C1, we rely on a data section base register (DSBR) and a global offset table (GOT) [3], as well as a form of position independent code specially enhanced for MMU-less systems. To meet condition C2, we devise 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 condition C3, we come up with quasi-static linking. In the following subsections these mechanisms will be explained in detail.

3.1.1 Position Independent Code for Condition C1.

Position independent code (PIC) is required for condition C1 because it can be executed from any location without modification. This is possible because PIC does not use absolute addresses when referring to symbols. PIC is used in most shared libraries, and a number of widely available compilers including the `gcc` are able to generate it. Since the addressing scheme used in PIC is closely related to other parts of our shared library scheme, we first explain in detail the PIC with MMU supports.

In PIC generated by `gcc`, symbols can be generally classified into three categories depending on how their

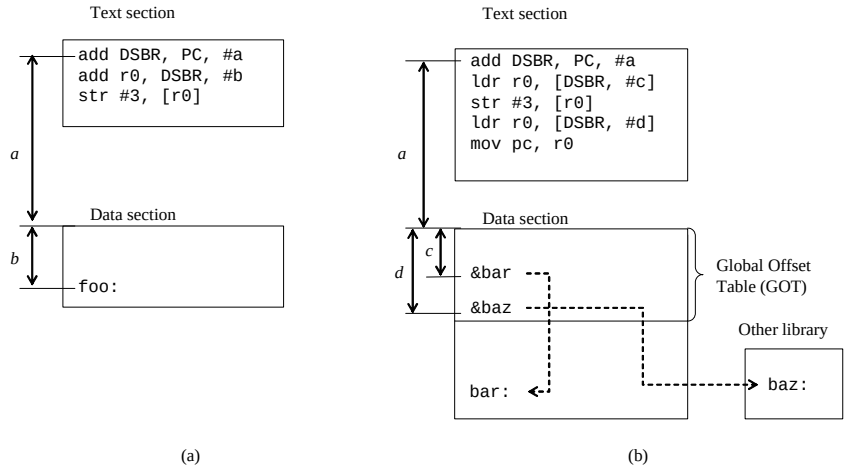

Figure 1. Pseudo code for referencing symbols in MMU-equipped systems: (a) writing an integer constant to local static variable `foo` using PC-relative addressing and (b) writing an integer constant to global variable `bar` defined in the same library and then jumping to global function `baz` defined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

addresses are dealt with: (1) local static functions, (2) local static data, and (3) global data and functions. Referencing local static functions is fairly trivial since they are contained within the same text section as the calling function. It 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 code section as shown in Figure 1.(a). Since this distance, denoted as *a* in Figure 1.(a), is fixed at linking time, the data section can be located in a PC-relative manner. The base address of the data section is loaded into a register called the data section base register (DSBR). Individual data symbols can then be located by adding an offset, denoted as *b* in Figure 1.(a), to the DSBR. On the other hand, referencing global data and functions is done through a global offset table (GOT) as shown in Figure 1.(b). The GOT is a table that contains entries for all of global symbols referenced in a library. The absolute address of a global symbol is determined and recorded by the loader at loading-time. In order to refer to a global symbol later on, the 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 Figure 1.(b). The GOT is located at a fixed offset in the data section, usually at its beginning. Thus, once the data section of the library is located, both global symbols and local data can be referenced. In MMU-equipped systems, the compiler is responsible for generating an instruction to load the DSBR with the data section's base address.

Unlike the addressing schemes explained above, in MMU-less systems, the data section of a library and thus its GOT are located at a unknown distance away from its code section. Therefore, PC-relative addressing cannot be used to load DSBR. In our scheme, loading the DSBR is handled differently by the loader and compiler. Specifically, the DSBR is initialized by the operating system's loader when a process is loaded for execution. In doing so, the loader determines the data section base addresses of the shared libraries used in the process and provides them for the process. The compiler-generated code of the process updates DSBR later on using the loader-provided addresses when a global library function is invoked. This requires that the `uClinux` loader and the code generator of the `gcc` compiler should be modified. Also, application programs should be compiled with an option that designates one of general purpose registers as the DSBR. This is needed to guarantee that the DSBR is not overwritten by application programs. We explain in detail the initialization and update of DSBR in what follows.

3.1.2 Data Section Base Table for Condition C2. In a process that uses multiple libraries, there are multiple code and data sections, each of which is loaded at an arbitrary address. Thus, the DSBR of a process should point to different data sections as application-to-library or inter-library calls are invoked. This requires a

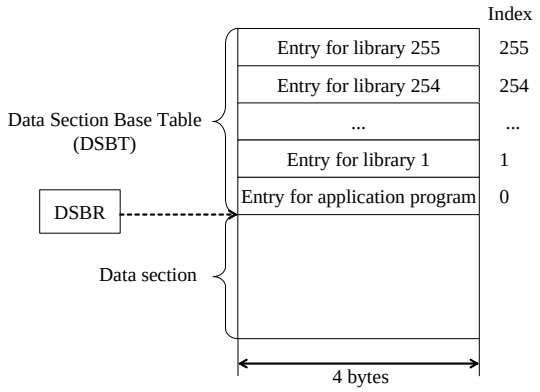

Figure 2. Structure of DSBT and method for accessing its entries.

runtime data structure called 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.

Figure 2 shows the structure of the DSBT. It has 256 entries, each of which is four bytes long. Each entry is dedicated to a specific shared library and contains the data section base address of that shared library. For this purpose, each shared library is given a unique numerical library ID, ranging from 1 to 255, with 0 reserved for the application program itself.

When a process is loaded into memory for execution, the loader allocates the text and data section, among others, of the process. It also allocates the data sections of all the shared libraries used in the process. Then, it creates the DSBT by filling up its entries with the allocated data section base addresses and attaches it to the front of the process's data section. It goes on replicating the DSBT and attaching its replica to the front of the data section of each shared library. Finally, it sets the DSBR to the data section base address of the process. We have modified the `uClinux`'s loader for this job. Figure 2 shows the configuration of DSBR and DSBT right after process loading. Under this scheme, 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 DSBT for all shared libraries at the expense of the increased memory space so as to allow a process to access both a data section and DSBT via a single DSBR regardless of the currently active data section. This way, we can avoid using an extra register for accessing DSBT. Note that general purpose registers are extremely valuable resource in enhancing the runtime performance of executable code. Moreover, the space overhead is relatively small in that the size of the DSBT can be configurable at build-time according to the number of shared libraries used by the system with a maximum of 255 shared libraries. For example, it would be 64 bytes per shared library if the number of shared libraries used in the system is 15, which is a relatively large number of libraries for a normal embedded system. Developers can even merge several distinct libraries at build-time to further reduce the table size.

During the execution, when an application-to-library or inter-library call is made, the DSBR should be updated so that it points to the data section base address of the called library. To do so, the process should first obtain the library ID of the called library and then compute 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 and set 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, we provide below the code for the `printf` function contained in the `uClibc` library. This code is generated by the modified `gcc` and the library ID of `uClibc` is 1. Note that the `gcc` compiler uses `sl` register as 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. Obviously, these additional instructions cause performance overhead whenever an inter-library call is made. In Section 4.2, we show that this has negligible impact on overall performance.

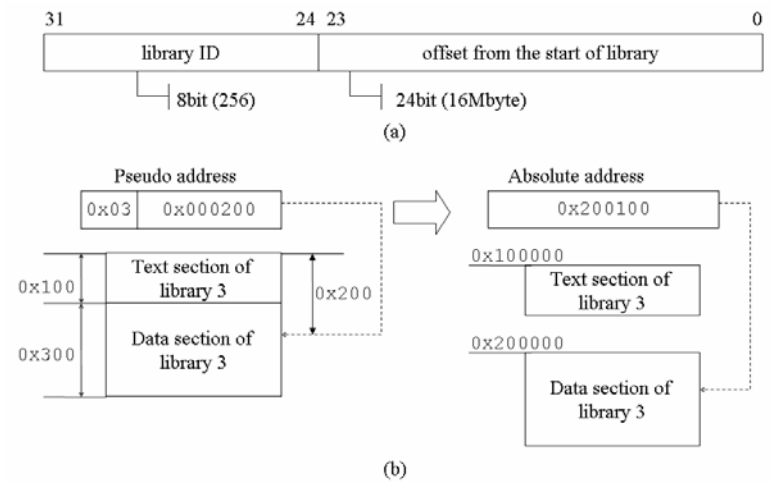

Figure 3. (a) Format of a pseudo-address and (b) an example of converting pseudo address `0x03000200` into an absolute address `0x200100`.

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

3.1.3 Quasi-Static Linking for Condition C3. Finally, to fulfill condition C3, we present the quasi-static linking mechanism. Quasi-static linking is a novel linking mechanism that statically links an application with a library using a logical numeric address for a symbol and dynamically determines the absolute address of a symbol at loading-time using a simple address resolution formula. It permits a shared library to be loaded at an arbitrary address since the permanent address of a symbol is dynamically determined and it avoids a symbol table at loading-time since symbols are statically bound anyway.

The two core components of the quasi-static linking mechanism are a pseudo-address and a pseudo-library. A pseudo-address is defined for a symbol 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 Figure 3.(a). The library ID and the offset are 8 bits and 24 bits wide, respectively. The offset is the distance from the start of the library to the definition of the symbol when the text section and the data section are contiguous.

A pseudo-library is a library that contains only a symbol table but not any code or data. Thus, a pseudo-library cannot be used at runtime to bind the library code and data into an application. Instead, a normal shared library object with the symbol table stripped off is used. Each entry in the symbol table specifies symbols by their names and their corresponding pseudo-addresses. As an example, Figure 4 depicts the symbol table for the `printf.o` object contained in the pseudo-library of the `uClibc` library. In this library, the pseudo-address of `printf`

```
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
```

Figure 4. Symbol table for *printf.o* in the pseudo-library of *uClibc*.

is defined as `0x01011ff8`: the function `printf` is defined in a shared library with the 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 the application, a feat possible because they have the same format as a static library. As a result, pseudo-addresses from the pseudo-libraries are embedded in locations in the application where absolute addresses of global symbols are expected. Also, relocation entries are created accordingly in the relocation table of the application for those locations since pseudo-addresses are required to be fixed at loading time. A relocation entry is just an offset into the application of those locations where the pseudo-addresses are used. It is important to mention that there is no need to maintain a symbol table in the shared library since symbols are already resolved 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 every 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. Note that the distance in memory between the text section and the data section should be added to this result if the offset is larger than the size of the text section, in other words, if the offset represents some location in the data section. Finally, the loader replaces the pseudo-address with the converted absolute address. Figure 3.(b) shows an example of extrapolating an absolute address from a pseudo-address.

Using quasi-static linking via pseudo-addresses makes the dynamic linking process simpler and faster than true dynamic linking with only a shared library. However, quasi-static linking carries the following restrictions, which we believe are not critical limitations for a small embedded system.

- Maximum number of libraries that can be in system is 255 (ID 0 is reserved for the application program).
- Library size (including text and data sections) cannot exceed 16MB.
- All application programs using a library should be re-linked with it whenever the library is updated because the offsets of the symbols inside the library might have changed.

Table II. Modified components and added functionality.

gcc	- Has functionality to get library ID value from command argument. - Adds instructions in prologue and epilogue of global functions. Instructions include 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 for their dependencies

3.2 Putting It All Together

Applying our proposed mechanisms to an embedded system involves modifying the `gcc` compiler [4] and the `uClinux`'s loader as explained above, as well as implementing the tool for creating pseudo-libraries. Table II summarizes the modified tool components and their added functions. When we developed a tool chain by integrating them, we put our emphasis on maintaining compatibility with conventional static libraries so that existing applications need not be rewritten. We explain how these mechanisms and corresponding tools are integrated coherently into the overall system development processes.

Figure 5 shows our new process for building applications and libraries, as well as the components that participate in each step. As shown in the figure, an application is built through exactly same process as when conventional static shared libraries are used. Its source code is compiled, and the generated object files are statically linked with pseudo-libraries to create an executable binary as explained in Section 3.1.3. The resultant executable, of course, is converted to BFLT (Binary FLaT file format) [6] by the `elf2flt` tool, since they should be executed on the target system.

However, the library building process is slightly different from that of the conventional static library scheme. As shown in Figure 5, our library building process accepts one or more library source files as its inputs and produces both a shared library and a pseudo-library as its outputs. In doing so, it temporarily generates a normal static library, since the tool creating the pseudo-library requires it as an 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 generates the object

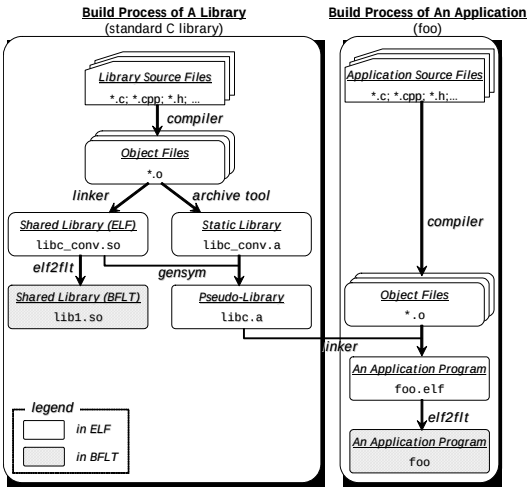

Figure 5. Development process of quasi-static shared library.

files in PIC. This compilation step is controlled by a command line option (`-mid-shared-library`) added to the original compiler. With this option, as explained in Section 3.1.2, the code fetching the data section base address to the DSBR is generated at the prologue of each global function, and the code restoring 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 the object files together to generate the shared library in ELF (Executable and Linking Format) [7]. The shared library is, in turn, transformed into BFLT so that it is loadable to the target system. Third, the archive utility (normally `ar`) collects the object files and produces the static library in ELF. As already explained, this is temporarily generated to create the pseudo library.

The final build step of the library build process is creating the pseudo-library. For this use, we developed a new tool named `gensym`. As depicted in Figure 5, `gensym` takes as inputs a static library and a shared library both in ELF. The pseudo-library is created by extracting the symbol tables from this static library and discarding the rest. Then, all symbols in the pseudo-library are 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 environment so that they can be quasi-statically linked with applications.

4 Experimental Evaluation

We have implemented the proposed quasi-static shared library scheme in a commercial ADSL home network gateway equipped with an MMU-less ARM7TDMI processor core, 2MB flash memory, and 16MB SDRAM. We have conducted a series of experiments to measure its memory usage and evaluate its performance overhead. Before presenting the measured results, we give an overview of the target applications and libraries.

The target system contains three classes of applications: system utilities such as `cp` and `mount`, modem utilities such as `adsldmt`, and communication utilities such as the `boa` web server. The applications use two libraries: the standard `C` library `uClibc` that is distributed with `uClinux` and the `SNMP` library `UCD-SNMP`. We applied the shared library scheme only to `uClibc` since UCD-SNMP is not shared among applications, but is used by only two `SNMP`-related utilities: `snmpd` and `snmptrap`.

4.1 Measuring Memory Usage

Since our overall aim is to reduce the power consumption and production cost of the system by

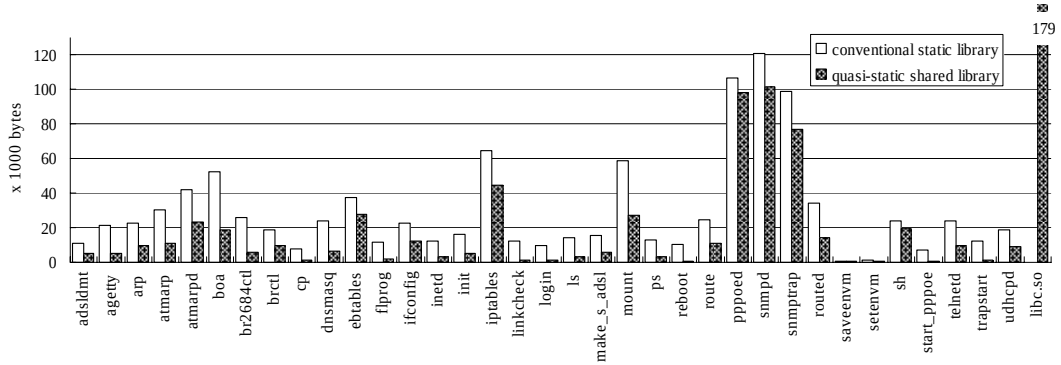

Figure 6. Flash memory usage.

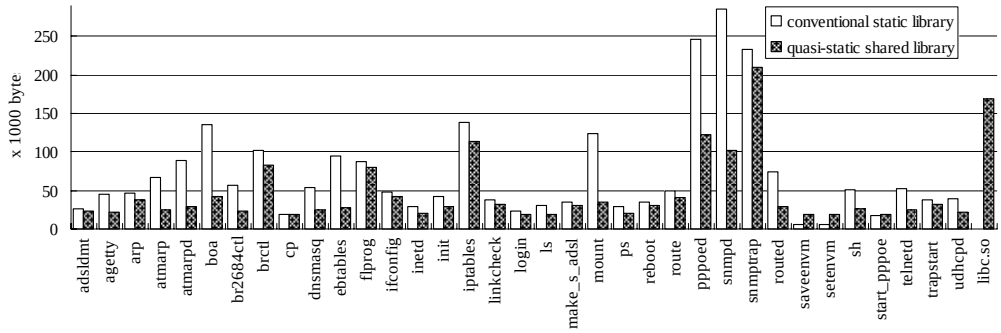

Figure 7. RAM usage.

reducing the amount of flash memory and RAM used, the metrics of our experiments are obvious: the amount of flash memory usage and RAM usage. While the former is defined for each application and library, the latter is defined for each application only. The flash memory usage of an application is defined as the sum of its code and data sections. The RAM usage of an application is defined according to which scheme is being used. In a statically linked application, RAM usage is defined as the sum of the code, data and bss sections. But if the quasi-static shared library scheme is used, we must also include the size of the libraries' data in calculating RAM usage. In calculating memory usage, we ignore the amount of memory consumed by relocation tables, headers of executable files, heaps, and stacks since they are constant regardless of library schemes.

Figure 6 depicts the changes in flash memory usage after our shared library scheme is applied. It shows a 35% reduction in flash memory usage from 1004KB to 651KB. The reason behind this is clear: library code is not copied into the executable file of each application but is instead kept as a single shared file (`libc.so` in Figure 6).

Figure 7 depicts the changes in the RAM usage. Overall RAM usage has been reduced 10% from 2435KB to 2200KB. Compared to the decreases in the flash memory usage, this is not a large decrease. In fact, the amount of RAM usage in some applications actually increases. This happens because the data section of an entire library is allocated to a process when using a shared library, while only the data sections of specific library routines are allocated to a process when using a conventional static library. If the data section of an entire library is larger than the code and data that a certain application uses from the library, there is no advantage (in terms of RAM usage) to applying quasi-static linking to that application, and this is the reason why the RAM usages of some applications are increased with the quasi-static shared libraries. In our scheme, it is possible for developers to selectively choose whether each application is linked to quasi-static shared libraries or conventional static libraries.

4.2 Measuring Performance Overhead

The target system, an ADSL home network gateway, has two operational scenarios: the normal operational scenario in which the target system performs protocol conversions from ATM traffic to Ethernet traffic and vice versa, and 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 scheme does not affect the performance of the target system, because all the protocol conversions are performed by the kernel, not by any user programs. We therefore turn our attention to the configuration scenario.

In order to evaluate the performance penalty caused by our schemes in the configuration scenario, we measured the response time for configuring the target system. In our system, a user configures the gateway using a web browser. The configuration request is serviced by the `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 (8 times for `sh`, 7 times for `ifconfig`, and 1 time for `iptables`). Table III shows the result with the response time representing the

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

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

time taken for the execution of `adsl.cgi`. The result shows that the performance penalty for the shared library is only 3.4 %.

5 Conclusions

In this paper, we have presented the quasi-static shared library scheme as a means of reducing both the dynamic and static memory requirements of heavily optimized, resource-constrained embedded systems lacking MMU. The proposed scheme is 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 schemes 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 as dynamic linking does. Avoiding symbol tables allows our scheme 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 scheme maintains compatibility with conventional shared libraries so that existing applications need not be rewritten. These characteristics make our scheme appropriate for COTS-based embedded systems that are subject to stringent performance and resource constraints.

We have realized our scheme by emulating the memory-mapping feature of MMU with a DSBR and DSBTs. This required us to modify the `gcc` code generator and `uClinux`'s loader and create a new tool named `gensym`. We have implemented the entire scheme in a commercial ADSL home network gateway and created a tool chain for building and deploying shared libraries. We have performed extensive measurements to analyze the RAM and Flash memory requirements of the home network gateway and to evaluate its performance overhead. The results are drastic: a 35% reduction in flash memory usage and a 10% reduction in RAM usage. These results were achieved with only a negligible performance penalty of less than 4%.

There are two research directions along which our scheme 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 scheme, will help to 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 scheme should be feasible for the vast majority of embedded systems. The second direction for our future work is to extend our framework to support eXecution-In-Place (XIP) [12]. XIP is a technique whereby a program stored in ROM or flash memory is run directly from the location where it is stored. With XIP, the text section of a program does not have to be loaded into RAM. Thus, XIP can effectively reduce RAM usage. In order to execute an application program in place, it should operate regardless of its location and its text section should never be modified for the execution. Note that our shared libraries fulfill these conditions, meaning that the same mechanisms used in our shared library scheme are also very effective in implementing XIP. The results look promising.

6 References

[1] uClinux, "Embedded Linux and Microcontroller Project," URL: http://www.uClinux.org.

[2] BusyBox, URL: http://www.busybox.net/about.html.

[3] John R. Levine, "Linkers & Loaders," Morgan Kaufmann, 2000.

[4] gcc, "GNU Compiler Collection," URL: http://gcc.gnu.org.

[5] Cadenux, URL://www.ridgerun.com.

[6] uClinux, "BFLT: Binary FLAT file format," URL: http://www.uclinux.org.

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

[8] Marc Sabatella, "Issues in shard library design," In Proceedings of the Summer 1990 USENIX Conference, pages 11-23, Anaheim, CA, June 1990.

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

[10] James Q. Arnold, "Shared libraries on UNIX System V," In Proceedings of the USENIX 1986 Summer Conference, 1986.

[11] D. M. D. M. Beazley, B. D. Ward, and I. R. Cooke, "The Inside Story on Shared Libraries and Dynamic Loading," IEEE Computing in Science & Engineering Vol. 3, Issue 5 (Sep/Oct 2001): pp 90-97.

[12] Don Verneer, "eXecute-In-Place," Memory Card Magazine, March/April 1991.

[13] P.W. Agnew and A.S. Kellerman, "Distributed Multimedia," p. 144, Addison Wesley, ACM Press, 1996.