# Performance Characterization of Prelinking and Preloading for Embedded Systems *

Changhee Jung
Embedded Software Research Division
ETRI
Daejon, 305-700, Korea
chjung@etri.re.kr

Duk-Kyun Woo
Embedded Software Research Division
ETRI
Daejon, 305-700, Korea
dkwu@etri.re.kr

Kanghee Kim
Mobile Communication Division
Telecommunication Network Business
Samsung Electronics Co., LTD.
kang.hee.kim@samsung.com

Sung-Soo Lim
School of Computer Science
Kookmin University
Seongbuk-gu, Seoul, Korea 136-702
sslim@kookmin.ac.kr

## ABSTRACT

Application launching times in embedded systems are more crucial than in general-purpose systems since the response times of embedded applications are significantly affected by the launching times. As general-purpose operating systems are increasingly used in embedded systems, reducing application launching times are one of the most influential factors for performance improvement. In order to reduce the application launching times, three factors should be considered at the same time: relocation time, symbol resolution time, and binary loading time. In this paper, we propose a new application execution model using a combination of prelinking and preloading to reduce the relocation, symbol resolution, and binary load overheads at the same time. Such application execution model is realized using *fork and dlopen* execution model instead of traditional *fork and exec* execution model. We evaluate the performance effect of the proposed *fork and dlopen* application execution model on a Linux-based embedded system using XScale processor. By applying the proposed application execution model using both prelinking and preloading, the application launching times are reduced up to 71% and relocation counts are reduced up to 91% in the benchmark programs we used.

*This work was supported in part by the IT R&D program of MIC/IITA [2006-S-038-02, Development of Device-Adaptive Embedded Operating System for Mobile Convergence Computing], in part by No. 379 from the Basic Research Program of KOSEF, and in part by the MIC, Korea under the ITRC (Information Technology Research Center) support program supervised by the IITA (IITA-2006-C1090-0603-0045).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*EMSOFT'07,* September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-825-1/07/0009 ...$5.00.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, run-time environment*; D.4 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

## General Terms

Design, Performance

## Keywords

embedded systems, prelinking, preloading, application launching

## 1. INTRODUCTION

Dynamically linked applications have been increasingly used in embedded systems as general purpose operating systems are adopted in embedded systems. As more diverse applications with significant sizes are installed for such embedded systems, the portion of dynamically linked applications increase accordingly. Therefore, the shared library processing times including relocation, symbol resolution, and library image loading steps have substantial impact on the overall application execution performance. For example, in [1], the GNOME startup time analysis results show that the most of the IO operations are due to shared library operations and around 65 % of non-contiguous IOs come from library accesses.

Application launching overhead is more crucial in embedded systems since the hardware performance of the embedded systems is not comparable to the performance of general purpose systems. Moreover, the prolonged launching times could not be accepted in user-interactive consumer electronics products since the response times of applications would

directly affect the values of the products (i.e., the launching time of HTML or WAP browser in mobile phones, the launching time of document viewer in PDAs and phones). Optimizing such application launching times through efficient handling of dynamically linked applications is necessary in the products.

In this paper, we propose an application execution model to reduce the handling cost of dynamically linked applications in launching applications: we perform relocation, symbol resolution, and shared library binary loading in advance before application execution. The prelinking (i.e., relocation and symbol resolution before application execution) and preloading (i.e., shared library binary loading before execution) techniques are combined in an application execution environment by modification of the existing dynamic linker. In order to apply the combined techniques to the existing general purpose OS-based systems, the *fork and dlopen* model is used for application execution instead of *fork and exec* model. In the *fork and dlopen* model, the applications are launched as shared objects by an application launcher and the shared libraries needed to execute the applications are prelinked and preloaded by the launcher.

The *fork and dlopen* application execution model with prelinking and preloading naturally gives a number of restrictions in launching and running applications. First, as we mentioned above, the applications need to be changed to shared objects and launched through *dlopen*. Second, most of the applications in a system need to be covered by predetermined set of shared libraries so that additional shared library loading would not be needed at run-time. Though the factors mentioned above seem to impair the flexibility of the system, the consumer electronics embedded systems could well accept the application execution model due to their characteristics. The consumer electronics products such as smartphones, portable media players, and set-top boxes normally have a centralized application launcher who controls the launching and termination of other applications. In addition, the shared libraries used by the applications could be well predetermined since most of the applications are fixed from factory configuration and additional applications would also utilize the already installed shared libraries. Moreover, once the products boot and start to run, the products are rarely rebooted, but typically woken-up from suspended modes. This would hide the launching time of the application launcher itself and preloading times of shared libraries since these are performed at boot time. Especially for mobile phones, the system boot is normally accompanied by subscriber registration process and thus the prolonged launching time would not affect the user perception.

We implemented the application execution model in a Linux-based embedded system with modification of dynamic linker in GCC. We evaluated the performance improvement of the *fork and dlopen* application execution model with the previous *fork and exec* model using a number of benchmark programs. Our experiments show that the launching times of applications are reduced up to 71 %, relocation counts up to 91 %, and the total execution times of applications are improved up to 11.3 % by using the combination of prelinking and preloading.

## 2. RELATED WORK

Efficiently linking programs has long been an important issue in operating systems area. Dynamic linking draws a number of issues in application execution performance. Efficient dynamic linking has been discussed and implemented in single-address-space operating systems such as Mungi [5, 3] and Nemesis [12] and the dynamic linking scheme has been used in Iguana project [4] which is a modified version of Mungi and used for commercialized systems. Most of the issues in dynamic linking in single address space have been discussed in [5, 3, 12] and those discussions could be well considered in general purpose operating systems. In this paper, we aim at devising and evaluating a method for efficient dynamic linking that could be practically used for general purpose operating systems such as Linux.

Prelinking [8] is an ELF image translation to speed up dynamic linking of ELF programs. It tries to remove symbol resolutions of ELF programs by incorporating the simple philosophy of the a.out binary format into the ELF format without compromising its flexibility. That is, it assigns a base address to which every shared library in the system should be mapped, and according to the base addresses, performs all symbol resolutions at link time. If there exists a case where the base address assigned to a prelinked shared library is already occupied by another shared library (e.g., when a non-prelinked library is already loaded to the base address), all symbol resolutions performed at link time for the prelinked library are cancelled at load time of the library, and the library is handled as a normal non-prelinked ELF library; that is, the base address is dynamically determined by the system, and the symbol resolutions occur during the run time of the program. To make an effective use of prelinking, it is important to cover as many shared libraries as possible in system design phase, thus reducing the possibility of non-prelinked libraries in system operating phase.

On the other hand, *fixed-up image caching* [11] is an operating system level technique developed in Spring OS to speed up dynamic linking. The basic idea is to maintain a cached copy of application executables and shared libraries, once the corresponding applications are launched and terminated. If an application has to be launched later, the cached copy of the associated executable and shared libraries is used. In this approach, it is important to assign a unique base address to each shared library, in order not to maintain multiple copies of a single shared library due to different base addresses. Since the approach requires large memory space for image caching, it is not adequate for embedded systems with no swap.

One final resort to completely avoid relocation, symbol resolutions, and library loading is to come back to static linking. In [2], static linking is revisited as an alternative to dynamic linking, which is combined with a notion called *message digests* to reduce the space overhead imposed by static linking in terms of both memory and disk. However, the space overhead of 40% relative to dynamic linking is too high for general purpose OS-based embedded systems we consider.

Recently, as general purpose OS-based embedded systems are increasingly used, there has been an effort to address the overheads of dynamic linking with a different application execution model. This is *quicklaunching* [13], which adopts the *fork and dlopen* model other than the traditional *fork and exec* model. In this approach, a process called launcher *preloads* all shared libraries that will be used in the system, while completing relocations and symbol resolutions. This

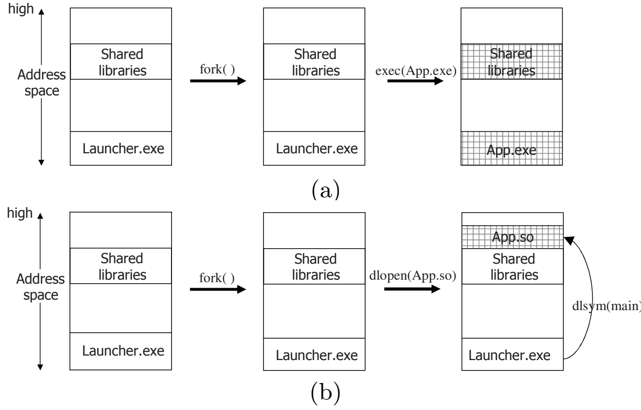


**Figure 1: (a) fork/exec model vs. (b) fork/dlopen model**

way the launcher can guarantee to each application no overheads due to the shared libraries, except for the application executable.

# 3. APPLICATION EXECUTION MODEL

In this section, we introduce the idea of preloading shared libraries, which starts to be used in general-purpose OS based embedded systems [13] but not quantitatively evaluated to the best of our knowledge. In such embedded systems, as more shared libraries of general-purpose systems are adopted unmodified, it becomes more crucial to speed up dynamic linking of ELF programs. Compared with prelinking, preloading is another complete mechanism to reduce the overheads due to dynamic linking by changing the application execution model from *fork and exec* to *fork and dlopen*. As can be seen in Section 4, preloading can give us a significant synergy, combined with prelinking.

## 3.1 fork-and-dlopen Execution Model

The traditional *fork and exec* model launches a new application in two steps: first, a parent process such as shells makes a child process by duplicating the address space via *fork*, and then the child process modifies its address space via *exec*. In the address space modification, the child process completely destroys all memory segments inherited from the parent process and maps a new ELF executable image and the required shared libraries into the empty address space. In this model, since loading of the shared libraries required by the child process, relocation and symbol resolution should occur for each single application, the application launching time could be significant. To reduce the application launching time, prelinking can be used in this environment to remove relocation and symbol resolution.

However, the application launching time can be better addressed by the *fork and dlopen* model. The essential problem of the *fork and exec* model is that the child process could not utilize the inherited memory segments from the parent process. If shared libraries of the child process are already loaded in the parent's address space, and all relevant relocation and symbol resolution are done in the parent context, the child process would incur no overheads upon execution due to the shared libraries. In this model, to preserve the inherited memory segments, the child process should launch the designated application image in form of an ELF shared object as well, instead of the ELF executable. Thus the child process loads the application image via *dlopen*, finds the address of *main* via *dlsym*, and finally executes the application by jumping to the address. To realize the *fork and dlopen* model, the set of shared libraries used in the system should be identified first, and be loaded into a common process called launcher. Figure 1 contrasts the two application execution model described above.

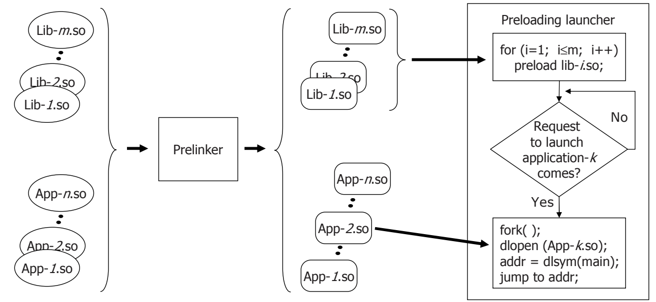


**Figure 2: The proposed framework**

## 3.2 Overall Framework

Figure 2 shows the overall framework of our application execution environment. We first identify all applications used in the system and build them as shared objects (App-1.so, App-2.so, ..., App-$n$.so). Next, for each application image, we find all the dependent shared libraries by recursively checking DT_NEED entries[1] in the dynamic section of the ELF binary and determine the whole set of shared libraries used in the system (Lib-1.so, Lib-2.so, ..., Lib-$m$.so). Next, we use the prelinker tool for both application binaries and shared libraries to remove relocation and symbol resolutions at run time of the applications.

With the above assumptions, we use the *fork and dlopen* model for our launcher program. The launcher program preloads all the shared libraries in advance, even if some of them are not actually needed for the launcher program itself. In a real embedded system such as smartphones, however, since the launcher program requires GUI interfaces like other GUI applications, many of the shared libraries are also utilized by the launcher program in practice. Next, the launcher program waits for a request of launching an application. When a user requests to execute an application $k$, the launcher forks a child process, and the child process invokes the application by loading the associated App-$k$.so binary.

It should be noted that each application invocation incurs almost no relocation and symbol resolution, since all application binaries and shared libraries are already prelinked. As we will see later, this leads to a significant performance improvement. Moreover, since all shared libraries used in the system are preloaded by the launcher process, we can obtain further performance improvement over the *fork and exec* model combined with prelinking.

Figure 3 shows the execution time decomposition of our application execution model. Our model can be described by four different phases:

1. launching phase of the launcher itself
2. preloading phase of the shared libraries

[1]Each entry contains the name of a dependent shared library

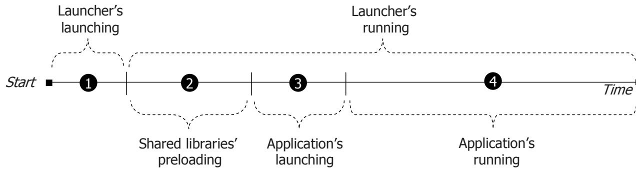


**Figure 3: Execution time decomposition**

| CPU | Intel PXA Bulverde 270 |
|---|---|
| Caches | 32KB L1 data, 32KB L1 instruction |
| SDRAM | 64 MB |
| NOR flash | 64 MB |
| Operating system | Linux 2.6.15 kernel |
| Dynamic linker | ld-2.3.2.so (glibc 2.3.2) |

**Table 1: Target embedded system specification**

3. launching phase of a target application
4. running phase of a target application

The first phase is launching phase of the launcher itself, which can be reduced by applying prelinking to the launcher's binary. The second phase is preloading phase of all the shared libraries used in the system, which performs all relocations and symbol resolutions in advance that would otherwise occur while each target application is running. It is important that the overhead of the preloading phase can be amortized as more and more applications are executed once the launcher executes. The third phase is launching phase of a target application, where it requires only relocation and symbol resolution of the application binary. The final phase is running phase of the target application.

Note that in our framework, it is also possible to consider the shared libraries that an application binary explicitly invokes via dlopen, which are not detected at prelink time. To do this, some other tool is needed for searching each application binary for such shared libraries and instrumenting the source code of the application so that the explicit dlopen call can be removed, but we do not consider such a tool in the paper.

## 3.3 Implementation

In order to practically use the preloading, we need to determine which software platform would be used. An operating system should provide the way of dynamic linking which allows an application to request the dynamic linker to load and link certain shared libraries at runtime. For this purpose, Linux and Windows support *dlopen* and *LoadLibrary*, respectively.

In addition, when converting static libraries to shared ones in order to exploit the preloading, special care should be taken to avoid runtime errors. Generally, when a shared object is compiled, only symbols are added by default to its dynamic symbol table that are actually used by some other modules. In other words, the dynamic symbol table of the shared object does not contain those symbols that are not referenced by other modules at compile time, even though they may be used at runtime using *dlopen()*. Therefore, undefined symbol errors should be inevitable when an attempt to refer such symbols is made at runtime, since the dynamic linker tries to find symbol entries that are not in the dynamic symbol table. To solve this problem, when compiling shared objects, a special compiler option should be specified that allows all symbols, not only used ones, to be added to the dynamic symbol table. For example, GCC (GNU Compiler Collection) supports a "-rdynamic" option for this purpose.

We implemented the prelinking/preloading-based application execution environment on a Linux-based embedded system. The system configuration is shown in Table 1. In order to augment the existing dynamic linking tool (*dlopen*) with the prelinking and preloading, the dynamic linker (*ld-2.3.2.so*) was modified. Our application execution environment assumes that relocation and symbol resolution process for prelinked shared libraries can be skipped by the dynamic linker when loading them with *dlopen()*. However, in current implementation of *ld-2.3.2.so*, such a mechanism works only for ELF executables instead of ELF shared objects. Therefore, we modified the dynamic linker so that it can selectively skip the process based on whether or not an ELF shared object is prelinked. Moreover, the dynamic linker was instrumented to measure performance metrics using software timers.

To apply prelinking to the target embedded system, Red Hat's *prelink* [8] was ported to the ARM architecture. We use it as a cross prelinker. And the preloading launcher was implemented from scratch in C.

# 4. EVALUATION AND DISCUSSION

In order to evaluate the performance improvement of our application execution model based on prelinking and preloading compared to previous application execution models, we have measured the launching times and total execution times of a number of benchmark programs. The measurement has been performed for both *fork and exec* and *fork and dlopen* program execution models explained in Section 3. In this section, the characteristics of the benchmark programs used in our experiments and the measurement results with the analysis are presented.

## 4.1 Benchmark Applications

We used eleven benchmark programs from three different benchmark suites MiBench [6], MediaBench [9], and GTKPerf [7]. We only choose programs where the number of used shared libraries is not less than 3. The descriptions of the benchmark programs are summarized in Table 2. For each benchmark program, the number of shared libraries used by each program and the portion of launching time in the execution time of the program are shown in the fourth and the last columns, respectively.

## 4.2 Execution Time

We present the measurements results of the proposed *fork and dlopen* application execution model and comparison with the results from the *fork and exec* application execution model. The comparison is performed in terms of *launching* time (Section 4.2.1) and total execution time (Section 4.2.2) in the following.

### 4.2.1 Launching Time

Table 3 shows the *launching* times of applications for the four different application execution models: Base and Prelink correspond to *fork and exec* application execution models: Base corresponds to non-prelinked *fork and exec* application execution model while Prelink to prelinked *fork and exec*

| Application | Suite | Problem | # of dependent shared libraries | % application's launching time |
|---|---|---|---|---|
| *fft* | MiBench | Fast fourier transform computation | 4 | 7.35% |
| *gsm.enc* | MediaBench | Rate speech transcoding coder based on the European GSM standard (encoder/decoder) | 4 | 6.28% |
| *gsm.dec* | | | 4 | 10.54% |
| *gtkperf* | gtkperf.sourceforge.net | Testing platform to run GTK+ widgets | 25 | 2.39% |
| *jpeg.enc* | MediaBench | JPEG encoder/decoder | 4 | 21.89% |
| *jpeg.dec* | | | 4 | 41.92% |
| *lame* | MiBench | MP3 encoder/decoder | 5 | 0.43% |
| *mad* | | | 7 | 19.86% |
| *mpeg2.enc* | MediaBench | MPEG2 video encoder/decoder | 3 | 0.29% |
| *mpeg2.dec* | | | 3 | 1.01% |
| *stringsearch* | MiBench | String search using a case comparison algorithm | 3 | 64.35% |

**Table 2: Benchmark applications used**

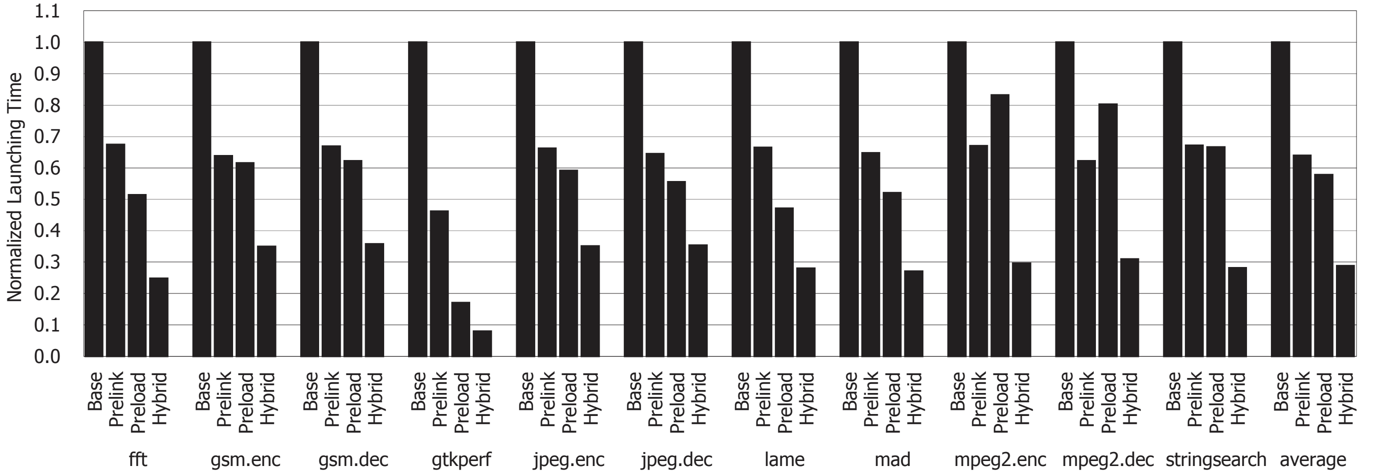


**Figure 4: Launching time of the applications with different schemes**

model. On the other hand, Preload and Hybrid correspond to *fork and dlopen* application execution models: Preload corresponds to non-prelinked *fork and dlopen* model while Hybrid to prelinked *fork and dlopen* model. In Preload and Hybrid models, the symbol resolution overheads for shared libraries are eliminated in advance as in the cases of Prelink model. Therefore, the difference between Preload and Hybrid would be the symbol manipulation overheads for application binary loading. Figure 4 shows the application launching times normalized to the Base execution model.

In *fork and exec* model, Prelink reduces the *launching* time up to 53 % and 36 % on average compared to Base model. This launching time reduction is caused by elimination of relocation time and symbol resolution time at launching phase.

In *fork and dlopen* model, Preload reduces the launching time by 42 % on average compared to Base model. This reduction is caused by elimination of binary loading overhead at launching phase. While Preload model performs better than Base and Prelink models for most of the benchmark programs, exceptions occur for `mpeg2.enc` and `mpeg2.dec` benchmark programs. The discussion on the reason why the programs show worse performance in Preload model than in Prelink model is described in Section 4.3. Hybrid model performs the best since the model combines both prelinking and preloading at the same time. As a result, Hybrid achieves a 71% average reduction in the *launching* time. In addition, Hybrid achieves 50% better performance on average than Preload and this improvement is caused by the elimination of relocation and symbol manipulation overheads.

### 4.2.2 Total Execution Time

Figure 5 shows the total execution times of the benchmark programs; the total execution time includes the launching time and the running time of each benchmark. Each bar shows the total execution time where the portion of the launching time is separately shown. Since the main objective of our application execution model is to reduce the launching time of each application, the total execution time reduction of each benchmark program is highly dependent upon the portion of the launching time in the total execution time (e.g., **`jpeg.enc, jpeg.dec, mad`**, and **`stringsearch`**). As shown in the figure, the running time portion of each application is little affected. The possibility to further reduce the *running* time of each application through *fork and dlopen* model is two fold; (1) relocation and symbol resolution caused by lazy binding [10] and (2) relocation, symbol resolution, and binary loading caused by *dlopen()* calls while an application is running. Unfortunately, even if it is possible, the time portion in the *running* time is quite small. That's why the *running* time is not reduced for almost all application. Especially for `gtkperf`, Prelink, Preload,

| Application | Base | Prelink | Preload | Hybrid | Application | Base | Prelink | Preload | Hybrid |
|---|---|---|---|---|---|---|---|---|---|
| *fft* | 23582 | 15893 | 12122 | 5841 | *gsm.enc* | 23584 | 15036 | 14517 | 8250 |
| *gsm.dec* | 23685 | 15838 | 14726 | 8453 | *gtkperf* | 130187 | 60114 | 22319 | 10444 |
| *jpeg.enc* | 23133 | 15322 | 13687 | 8107 | *jpeg.dec* | 23942 | 15435 | 13306 | 8469 |
| *lame* | 28632 | 19047 | 13491 | 8030 | *mad* | 34720 | 22499 | 18059 | 9417 |
| *mpeg2.enc* | 20673 | 13854 | 17198 | 6130 | *mpeg2.dec* | 21278 | 13243 | 17083 | 6590 |
| *stringsearch* | 20318 | 13651 | 13527 | 5720 | *average* | 33973 | 19994 | 15458 | 7768 |

**Table 3: Launching time of applications (absolute values)**

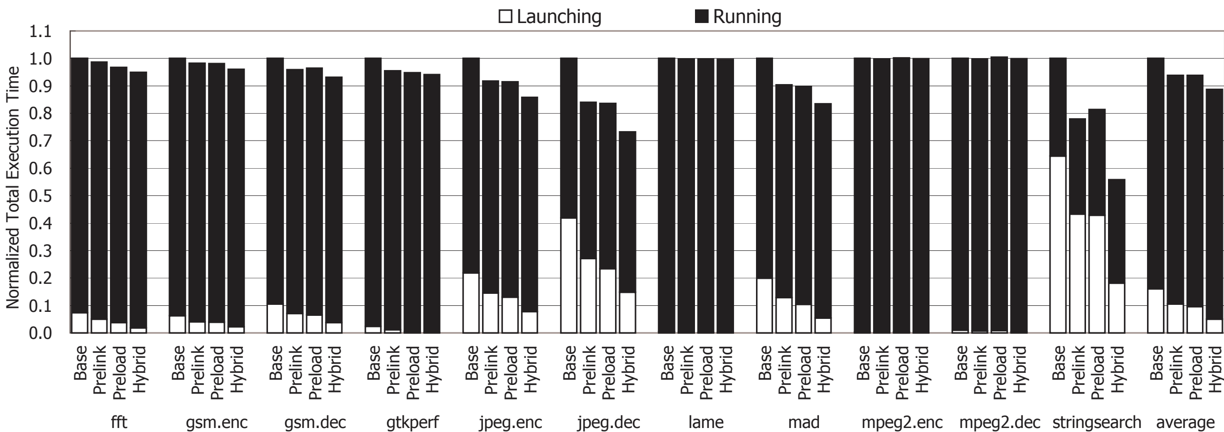


**Figure 5: Total execution time of the applications with different schemes**

and Hybrid reduce the *running* time by 3.27%, 3.31%, and, 3.75%, respectively. This is due to the characteristic of `gtkperf` which loads a number of shared libraries (1858) using *dlopen()* at runtime.

On average, Prelink, Preload, and Hybrid reduce the total execution time by 6.1%, 6.2%, and 11.3%, respectively. That results mainly from reduced *launching* time.

## 4.3 Relocation Counts

The performance improvement by prelinking and preloading would be tightly correlated with the relocation counts of the applications during their launching and running. Figure 6 shows the relocation counts of the benchmark programs for the four application execution models. The relocation counts are shown normalized to Base model. The white portion of the relocation counts corresponds to those performed at application launching phase while the black portion to the relocation counts at running phase. The figure shows that the relocation count of Prelink model is zero for almost all applications except for `gtkperf`. This is due to the runtime characteristic of `gtkperf` that loads shared libraries whose dependency information does not exist in the dynamic section of its executable file; prelinking works only for the shared libraries found in DT_NEED entries in the dynamic section of ELF binaries (see Section 3.2).

Preload reduces the relocation counts significantly for all the applications compared to Base model. The difference of the relocation counts reduction between Preload and Prelink is due to the relocation and symbol resolution overheads of the application itself since Preload does not use prelinked binary image for the application itself. The resultant relocation counts reduction of Preload model reaches 59 % on average. The actual relocation and symbol binding of the application image is typically performed at running phase through lazy binding [10]. Eliminating such run-time relocation and symbol binding overheads could be possible by combining Prelink model and Preload model. Preload model shows exceptionally poor performance for `mpeg2.enc` and `mpeg2.dec` benchmark programs in the aspect of relocation counts. The relocation counts for those benchmark programs are even quite larger than the relocation counts in Base models.

In `mpeg2.enc` and `mpeg2.dec`, there are especially many global symbols. Recall that the application itself is changed to a shared object in Preload model to use *fork and dlopen* model, that is, the application shared object is also position-independent code (PIC) [10] like the shared libraries it depend on. Thus the global symbols used in the application shared object cannot be resolved before runtime, since their real addresses are not determined until the dynamic linker maps them. Therefore, additional relocations and symbol resolutions could occur. Such additional relocation counts are presented in the figure and reach up to 37 %. In other words, the additional relocation counts for `mpeg2.enc` and `mpeg2.dec` benchmark programs at running phase are due to lazy binding of the procedure addresses used in the application while the additional relocation counts at launching phase are due to the the other global symbol resolutions; the global symbols would be mostly data used during encoding and decoding algorithms. The combination of prelinking and preloading could eliminate the additional symbol resolution overheads since the prelinking solves the run-time global symbol resolution problem. Hybrid model eliminates such relocation overheads as shown in Figure 6. Hybrid has just one relocation for *main()* in all applications and all other relocations are handled by prelinking.

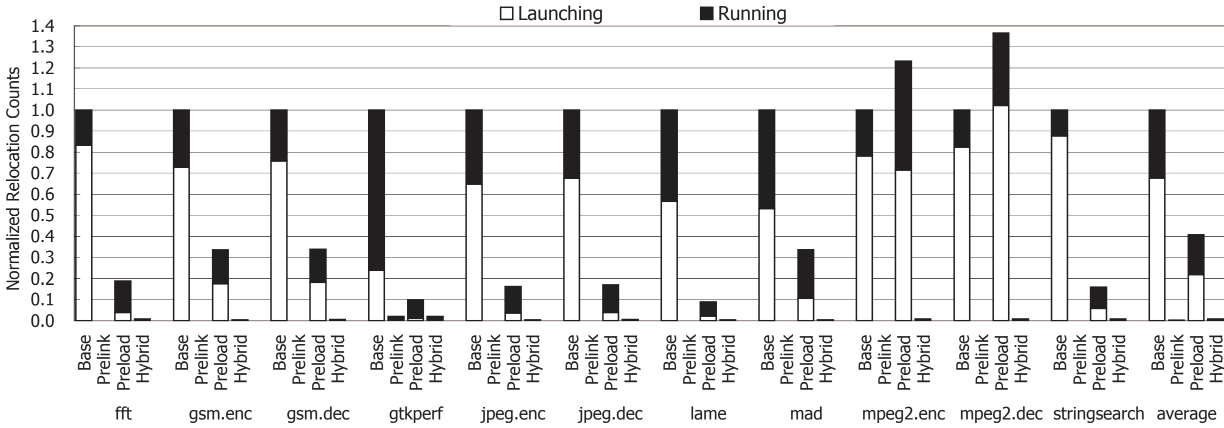


**Figure 6: Relocation count of the applications with different schemes.**

## 4.4 Preloading Time

Final evaluation for our application execution model is to compare preloading times between Preload model and Hybrid model. This experiment is to see the impact of the prelinking on the preloading time of shared libraries. Since Hybrid model uses prelinked binary, the preloading overhead is much less than the overhead in Preload model. The preloading time improvement is significant as the number of shared libraries needed to be loaded increases. Therefore, it is meaningful to see how much improvement in preloading time is achieved as the number of shared libraries to be loaded increases. To obtain the results on the preloading time improvement effect by prelinking, we evaluate the *preloading* time when the number of the shared libraries is varied from 10 to 100. Figure 7 compares the *preloading* time for the Preload and Hybrid models, normalized to the Preload model. The x-axis represents the number of the shared libraries loaded into the launcher. The figure shows that the impact of the *preloading* time reduction due to Hybrid becomes increasingly significant (28-71%), as the number of the shared libraries increases.

## 4.5 Repeated Execution

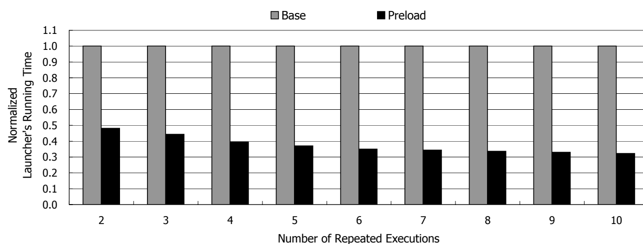


**Figure 8: Repeated execution effects**

The impact of preloading on the application launching time improvement would be increased significantly when an application is launched repeatedly in a system. Normally, in consumer electronics embedded systems, some applications are very frequently executed and thus the launching and running phases performance of the applications affect the overall system performance significantly. In order to quantify the impact of preloading on the application launching time improvement in repeated executions of a single application, we compare the execution time of stringsearch benchmark program between Base and Preload models in repeated executions. The figure shows that the increase of the repeated execution times causes a larger time difference between the two models. This reveals that the performance improvement in launching times of applications would increase significantly as the numbers of executions of the applications increase.

# 5. CONCLUSION

We have presented a new application execution model for general purpose OS-based embedded systems based on prelinking and preloading to reduce the launching times of applications. The proposed application execution model follows *fork and dlopen* instead of *fork and exec* model. The application execution model is implemented on an embedded system running Linux by modifying the existing dynamic linker.

The application execution model is categorized into four different configurations depending on the techniques used: Base for non-prelinked binary execution model for *fork and exec* model, Prelink for prelinked binary execution model for *fork and exec* model, Preload for non-prelinked binary execution model for *fork and dlopen* model, and Hybrid for prelinked binary execution model for *fork and dlopen* model. Performance comparison results for the four configurations show that Hybrid execution model improves application launching times up to 71 %, relocation counts reduction up to 91 %, and total execution times improvement of applications up to 11.3 %.

The results indicate that the application execution model using prelinking and preloading would be appropriate for consumer electronics embedded systems where predetermined set of shared libraries are commonly used, centralized application launcher controls launching and termination of applications, and the system rarely reboots.

# 6. REFERENCES


[1] L. Colitti. Analyzing and improving GNOME startup time. In *Proc. of the 5th System Administration and Network Engineering Conference*, Delft, The Netherlands, May 2006.

[2] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa. Slinky: Static Linking Reloaded. In *Proc. of*

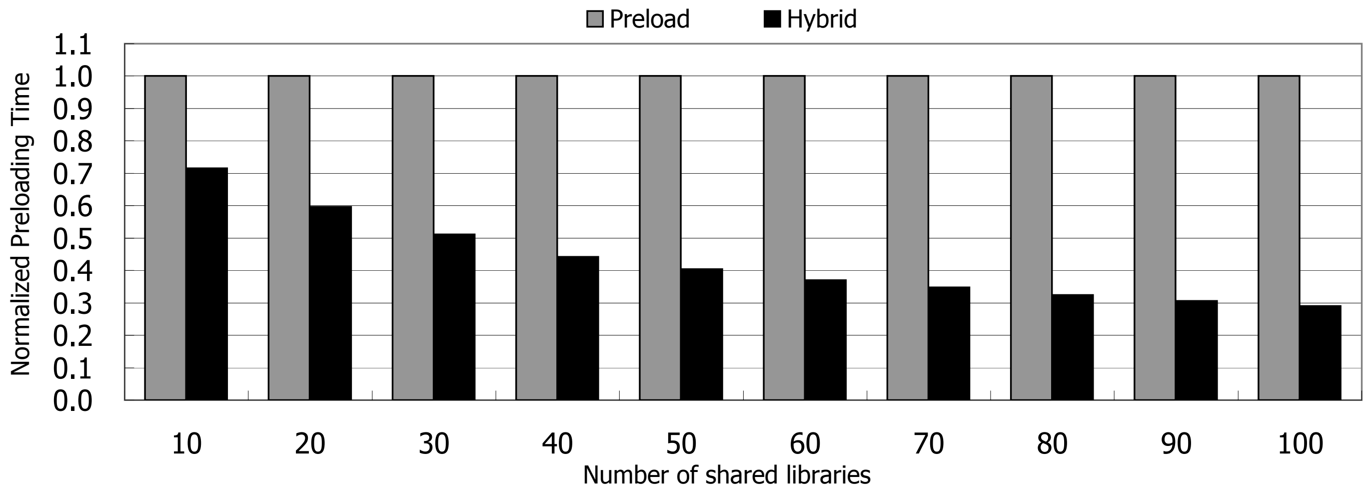


**Figure 7: Preloading time comparison of the original preloading and our application execution environment**


*the USENIX Annual Technical Conference*, Anaheim, CA, 2005.

[3] L. Deller and G. Heiser. Linking Programs in a Single Address Space. In *Proc. of the 1999 USENIX Technical Conference*, Monterey, CA, 1999.

[4] N. ERTOS. Iguana project, 2006. avaialable at http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/.

[5] J. V. S. R. J. L. Gernot Heiser, Kevin Elphinstone. The Mungi Single-Address-Space Operating System. *Software: Practices and Experiences*, 28(9), 1998.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, December 1998.

[7] http://gtkperf.sourceforge.net. Benchmarking Tools Designed to Test GTK+ Performance. 2005.

[8] J. Jelinek. Prelink. Technical report, Red Hat, Inc., 2004. available at http://people.redhat.com/jakub/prelink.pdf.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[10] J. R. Levine. *Linkers and Loaders*. Morgan-Kaufmann, 1999.

[11] M. N. Nelson and G. Hamilton. High Performance Dynamic Linking Through Caching. In *Proc. of the USENIX Summer 1993 Technical Conference*, Cincinnati, OH, 1993.

[12] T. Roscoe. The Structure of a Multi-Service Operating System. Technical report, University of Cambridge Computer Laboratory, 1995. TR-376.

[13] Trolltech. Developing Graphical System for Embedded Linux, 2006. available at http://www.freescale.com/files/abstract/overview/FTF2006_PC202.pdf.