Dynamic Loading in an Application Specific Embedded Operating System [⋆]

Stefan Beyer[1], Ken Mayes[2], and Brian Warboys[2]

[1] Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, 46022 Valencia, Spain
stefan@iti.upv.es
[2] Centre for Novel Computing, Department of Computer Science, University of Manchester, M13 9PL, United Kingdom
{ken, brian}@cs.man.ac.uk

Abstract. Traditionally, configuration of operating systems is done statically at compile- or link-time, but recently dynamic run-time configuration has become possible. Embedded systems however have constraints, such as limited memory and real-time requirements, that prevent many dynamically configurable operating systems from being used in an embedded system.
Dynamic configuration has associated limitations: either execution time overheads, due to complex code structures, or restricted flexibility. However, loading compiled code and linking it immediately at load-time avoids many of these overheads. This paper describes efficient dynamic loading and linking techniques employed as part of the Arena special-purpose operating system to allow embedded systems to be configured by replacing resource managers, such as the process manager. In Arena operating system managers reside in user-level libraries. A general-purpose loading-framework, designed specifically for embedded systems, is introduced and the result of performance experiments are presented.

1 Introduction

Embedded systems are special-purpose systems. They are often designed to perform very specialised tasks. However, recently embedded systems have become much more widespread and a high degree of flexibility is expected of systems with very limited resources. Any operating system running on such a system has to adapt to very specific application requirements. Therefore, configurable operating systems seem advantageous for embedded systems. There are two ways in which operating systems can be configured:

Static configuration is done at compile- or link-time. The operating system consists of components, which are combined to build a specialised "version" of the operating system.

[⋆] This work has been partially supported by the Spanish MCYT grant TIC2003-09420-C02-01.

Dynamic configuration is performed at run-time, either through external input, for example via user interaction, or automatically, as the result of the application requesting a re-configuration.

Static configuration tends to be more efficient at run-time, but it is less flexible than dynamic configuration. Static configuration is limited in that the type of specialisation needed may not be known until run-time. For instance, a memory management system can take advantage of information about page-usage collected at run-time to alter the page replacement policy, in order to reduce paging overhead.

This paper concentrates on the dynamic re-configuration of embedded systems. All the re-configurations performed in the experiments are application-driven. That is, the re-configurations happen as a reaction of the application to its current state. It is however easy to adapt the techniques described to perform re-configuration as a result of user-input.

The requirements for a dynamic configuration system for embedded operating systems are as follows:

Requirement 1: The system should allow low-level resource managers to be configured. This is important to allow the flexibility needed to adapt to the very different needs embedded applications might have.

Requirement 2: The run-time overhead should be minimal. Embedded systems are often designed to work with a very minimal hardware specification to save both cost and power. A dynamic configuration system should not introduce any significant overhead into the execution speed of the embedded application. Note, that this requirement does not refer to the speed of the actual reconfiguration, but to the run-time efficiency of the embedded application in general. Configuration efficiency is covered by requirement 5.

Requirement 3: The memory footprint should be small. Memory on embedded systems is limited. Particularly on systems without a Memory Management Unit (MMU), where no virtual memory is available, it is vital that the dynamic configuration system does not take up much memory.

Requirement 4: The system should not require an MMU. Many embedded systems do not have an MMU. It is on systems without MMU that dynamic configuration is most problematic.

Requirement 5: The Reconfiguration should be reasonably fast compared to the lifetime of a long-lived application. The more often a reconfiguration occurs, the faster it should be accomplished, to keep the impact on the efficiency of the embedded system to a minimum. For example, if a reconfiguration is to take place once every 10 minutes, it is not a problem if it takes 10ms. However, if a reconfiguration of such a duration was to take place every 20ms, the overhead would be huge.

Requirement 6: Real-time computing should be possible. The system should be predictable, so that real-time constraints can be met. Whether a system is in a "real-time mode" may depend on the particular configuration, but the introduction of a dynamic configuration system should not make real-time computing impossible.

Existing systems have been reviewed in the context of these requirements and have been found unsuitable. Therefore, a system which fulfils the requirements has been developed, based on dynamic code loading. The approach has been developed for operating system configuration on embedded systems with limited resources, but can also be used as the basis for a flexible component system at application-level.

The dynamic code-loading approach has been applied to the Arena library operating system, in which Operating System Mangers (OSMs) are implemented in user-level libraries, linked to the application. In this scheme, the application requests the loading or replacement of an OSM from a remote system over a network. A local dynamic linker then links the OSM into the running application. Note that this differs from dynamically-linked shared libraries [1], in that the loading and linking of the OSM happens during execution, rather than at application-load-time[3].

Two possible target applications have been implemented to demonstrate the flexibility of the system [2]. The first of these case studies is a system which allows the replacement of the process manager (PM). The application specifies a new required scheduling policy and the system loads an appropriate PM over the network and links it into the application. Performance experiments have shown that there is no measurable overhead in running a dynamically-linked PM, compared to a statically linked PM, once the loading and linking has been achieved [3].

As a second case study, network protocol loading has been investigated. A system has been implemented which automatically loads transport and application level protocols when needed. TCP [4] and HTTP [5] have been used as example protocols. The system saves valuable memory by “listening” to specific protocol ports without the full protocol implementation being present on the system. The protocol is loaded when a message needs to be sent or received. To demonstrate this loading of network protocols, a small embedded web server has been implemented. TCP and HTTP are only loaded when a request for a web page is received. A possible target application of this could be a multimedia device, which uses UDP [6] to stream data most of the time, but might occasionally be re-configured using a remote web interface over HTTP and TCP. Using dynamic protocol loading, it is possible to save memory otherwise occupied by these protocols for the actual streaming data.

This paper, rather than focusing on the two case studies, concentrates on the results of further experiments which have been performed to analyse the behaviour of the code loading system. To this end code modules from real world libraries have been loaded. The load times were measured and compared against code size, the number of relocation entries in the files and the number of symbols that had to be resolved.

[3] There are performance improving measures with dynamically-linked shared library approaches that delay the resolution of certain symbols until the first reference to them is made at run-time.

2 The Arena Operating System

The work described here is based on the Arena library operating system. Arena is an application-oriented operating system [7] [8] intended for both distributed and real-time applications [9] [10].

Arena introduces a separation between mechanism and policy. Low-level mechanisms are provided by a hardware-specific nano-kernel, the hardware object (HWO). The HWO provides mechanisms, such as the ability to save and restore the contents of the registers, but does not impose any policy on the context in which these registers are saved and restored. This architecture keeps the HWO free of operating system policy, but provides an architecture-independent hardware abstraction with opaque data types for low-level entities, such as register contexts.

All operating system policies are implemented in user-level libraries. These libraries are linked to the application, allowing the application programmer to choose the policies required, by linking to different version of these libraries.

The tight coupling between application and operating system policies leads to operating system managers (OSMs), such as the process manager, residing at user-level. The OSMs interact with the HWO through the HWO provided downcall interface. Furthermore, OSMs provide an upcall interface to enable the HWO to cause OSM code to be executed.

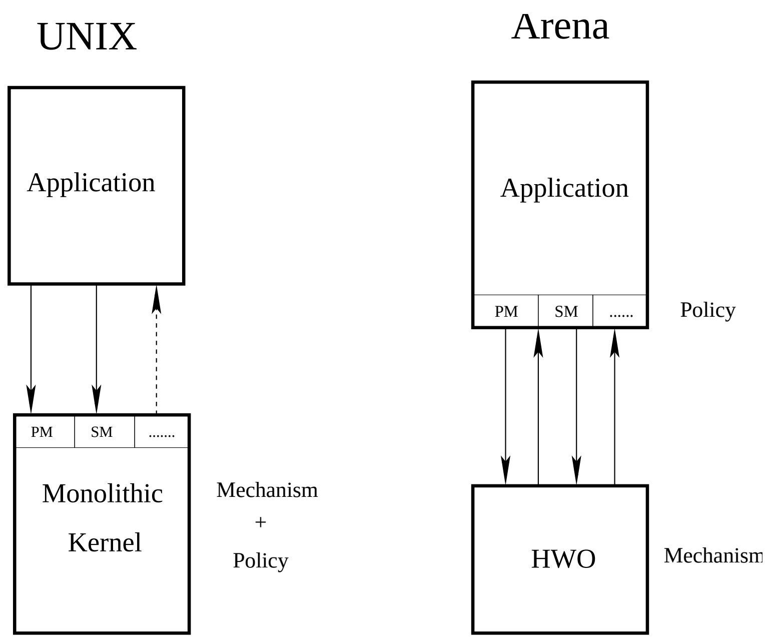

Fig. 1. Arena vs. Unix

Figure 1 compares Arena to Unix. Whereas in Unix the policies are general-purpose and are contained within the monolithic kernel, in Arena they are linked to the application. Therefore, it is much easier for an application to modify a policy to match its requirements. This makes Arena "application-compliant". As noted above, Arena provides an upcall interface, to allow execution of application-specific policies when low-level events occur. It can be argued that Unix provides an upcall interface in the form of Unix signals, but in Arena upcalls are used as the default mechanism to provide application-compliant event handling. On the occurrence of a hardware event, the HWO can make an upcall to some user-level resource manager. The upcall mechanism enables deferred processing of the event via an application-specific event handler thread. Fig. 2 shows how a hardware interrupt may cause the HWO to invoke the user-level process manager, which schedules a user-level event handler thread.

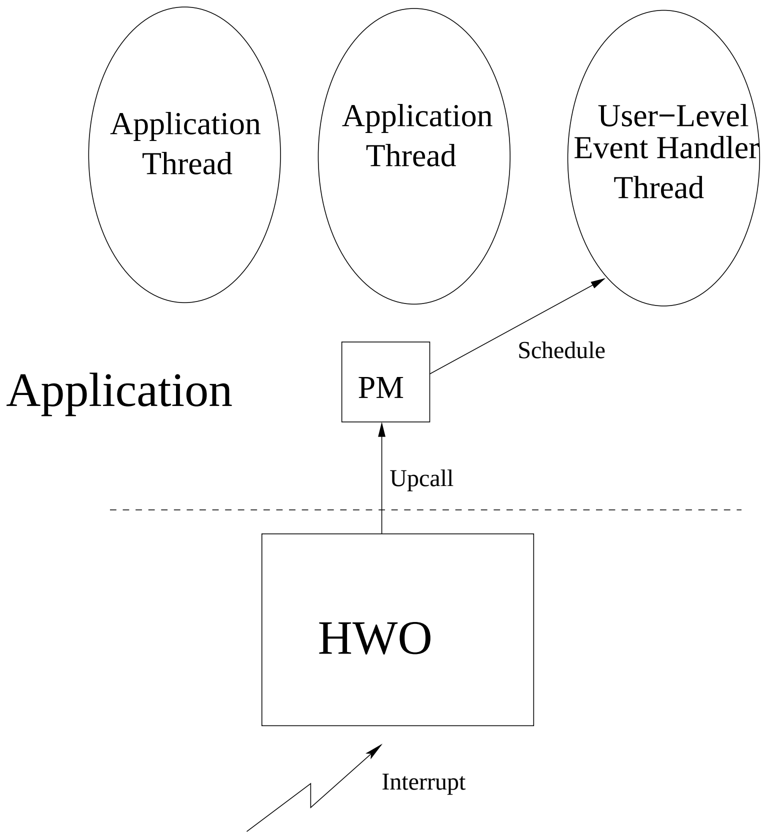

Fig. 2. Arena Event Handling

3 Dynamic Code Loading

OSMs are implemented as user-level resource managers in libraries in Arena. Formerly, these were statically-linked to the application to achieve re-configuration. It is a logical development to allow the application to load different versions of these libraries dynamically as a means of re-configuring the operating system. This work introduces the dynamic loading and replacement of user-level modules which can be either regular code modules or resource managers on Arena. In contrast to other dynamic configuration systems, the system introduced here, is optimised for embedded systems, especially embedded systems without an MMU and without secondary storage. In order to reduce the run-time overhead, code is patched directly, in the same way as performed by a static linker.

The dynamic code loading system described in this paper provides a library at the application level which can be used to load any piece of code. An application programmer could use this library to directly load or replace operating system components. However, the application programmer is assumed not to be an expert in operating systems implementations. Furthermore, he might not be the ideal person to make the choices to configure the system. The library provided is aimed at operating system and system level programmers for the implementation of re-configuration layers. Such a re-configuration layer provides certain options for the application programmer to choose from, such as “real-time mode” as a scheduling policy. The layer then makes the necessary choices to load an appropriate scheduler. The actual module loading is hidden from the application programmer. Two such re-configuration layers have been implemented as part of the case studies mentioned in Section 1.

4 Previous Work

4.1 Dynamically Configurable Operating Systems

Many conventional monolithic operating systems allow modules to be loaded into a running kernel. Linux and its kernel module loader [11][12] are a readily available example. Conventional micro-kernel-based systems, such as Mach [13], place OSMs in user-level servers. An OSM can theoretically be replaced by stopping a server and restarting a different version of it. However, these systems tend to be general-purpose and cannot give full control to applications, due to their multi-application and multi-user paradigms. Another problem is the fact that certain low-level policies, for instance in scheduling, cannot be modified. These systems violate requirements 1 and 6.

The Kernel Toolkit (KTK) [14] and Chimera [15] are systems that consist of a selection of configurable components, which have to be present on the system all the time, meaning that the system might be relatively large, if high flexibility is required. Therefore, there seems to be a trade off between requirements 1 and 3 in these systems.

Systems based on scripting (μChoices [16]), type and pointer-safe kernel extensions (Spin [17]) or virtual machines (Inferno Operating System [18], Java

[19]) do not allow configuration of certain low-level resource managers and therefore violate requirement 1, with some of them violating other requirements as well.

4.2 Dynamic Code Loading Systems

Distributed systems, such as CORBA [20] or Jini [21] "emulate" dynamic code loading. However, the network latency of service access might be unacceptable for some real-time applications (requirement 6). Most importantly, such distributed approaches do not allow low-level system manipulation (requirement 1).

Dyninst [22] is a somewhat low-level approach to dynamic code loading. It lacks flexibility, as it cannot link in arbitrary code (requirement 1).

Probably the most suitable approaches for arbitrary dynamic code loading are based on dynamic linking.

ELF systems [23] typically provide an API to the dynamic linker that can be used by the programmer to implicitly load executables. Apart from relying heavily on a UNIX environment, this system uses ELF shared objects, which are used for shared libraries. These shared libraries are loaded through the memory management subsystem on UNIX systems and rely heavily on the fact that pages are only loaded when needed (requirement 4). Therefore, the components of a library tend to be combined in a few big shared object files and it is not trivial to extract smaller sized-objects from the shared objects, such as relocatable object files from static library archives.

DLD [24] enhances a.out-based systems with dynamic loading and unloading of modules. DLD is a library package providing the ability to load relocatable object files, normally used as input files for static linkers, into a running application. The unlinking process relies on a garbage collector. DLD is the closest of all existing systems surveyed to the loading system described here. However, it was designed for UNIX systems and certain aspects of it, in particular the use of a garbage collector, make it less useful for embedded systems with memory restrictions and real-time constraints (requirement 6).

5 The Dynamic Object Loader

A dynamic object loader (DOL) has been developed for the Arena operating system. Figure 3 shows a overview of Arena with the DOL and a process manager switcher (PMS). The PMS is described in [2] [3]. In the Arena HWO nano-kernel, the Arena loader protocol (ALP), a very light weight transfer protocol, resides at the top of the network protocol stack. ALP is similar to TFTP [25] and is implemented directly on IP [26] in the prototype implementation. It provides the DOL with a simple send and receive interface, which allows the transfer of modules (MOD in figure 3) from a remote module server. The remote system contains an *application server*, which answers requests for whole applications and a *module server*, which is responsible for the transfer of modules (figure 4). ALP packet types allow requests for either whole applications, whole modules or individual symbol or string tables. This ALP interface is used by the DOL, which is linked into the application at user-level to load the modules and link them into the application. Loadable modules are contained in ELF relocatable object files.

The DOL can be used by the application either directly or through a special OSM layer, such as the PMS in figure 3. The application is loaded by the HWO using its application loader component ("Appl.Load" in figure 3). This application loader interacts with the remote application server to pull over the application executable. Once the application has been loaded in memory, it may require the loading of further modules. That is, subsequently, as required, modules can be loaded by the DOL. The DOL interacts with the remote module server to pull over the required modules.

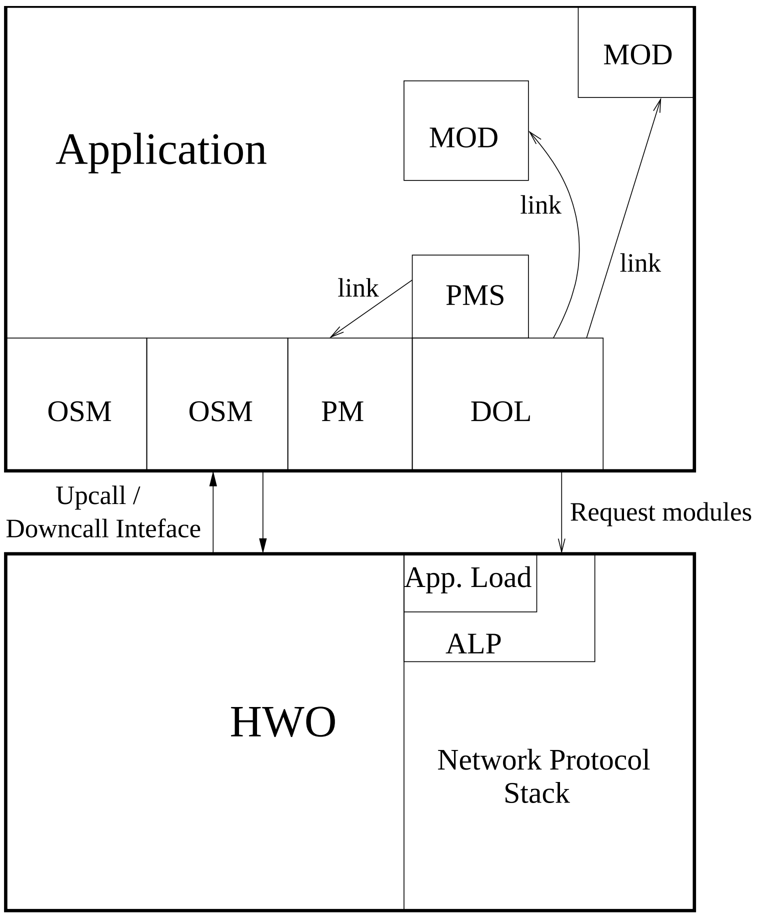

Fig. 3. System Overview

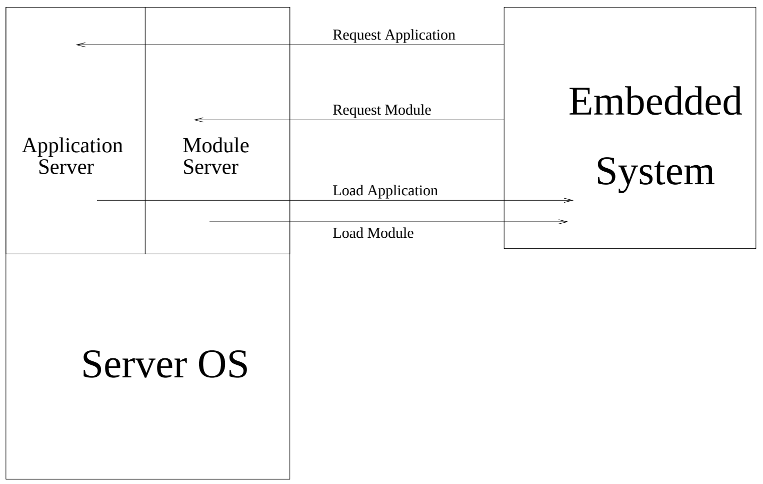

Fig. 4. Application and Module Server

It is vital that the DOL keeps a track of the symbols and string tables of the main program and of loaded modules so that symbols can be resolved and linked. In order to achieve this, the DOL maintains state with an entry for each loaded module. A module entry in this state contains the name of the module, the locations and sizes of the symbol and string tables and information about all the sections of the module [4]. Each module is also given a type. For example, regular modules (i.e.non-OSM modules) are of type `REG` and process managers are of type `PM`. The main program also has an entry in this module state, of type `PSEUDO`, so that symbol references to the main application can be resolved.

The initialisation of this DOL state is achieved by a call to

```
int dol_init (char *name);
```

`dol_init` takes the name of the main application as an argument. Its main purpose is to set up the PSEUDO entry in the DOL state. The remote module server is contacted and the string and symbol tables of the main application are requested. Since the module server executes in the same context as the application server (which sent the application itself to Appl.load), the module server can obtain the required string and symbol tables and send them to the DOL. `dol_init` then creates the PSEUDO module entry in its state. The main application symbols and strings are now accessible to the DOL.

When a module is required the function

```
int dol_load_module (char *name, int type);
```

[4] Not all sections of the ELF relocatable file containing the module have to be loaded.

is called. This loads the specified module into application memory and updates the DOL state with a new entry for the new module. The module server is contacted with a request for the section header table and section header string table. The DOL loads all loadable sections, including the string and symbol tables. The DOL state for this module is set to the specified type, and the locations of the string symbol tables noted. Next, the symbol table is *relocated* to contain the actual location of each symbol declared inside the module. This is followed by looking for sections in the newly-loaded module containing relocation information and performing each relocation. References which cannot be resolved within the module itself are undefined references, and require DOL to search through its module state for the location in other, previously loaded, modules. These undefined references are resolved by patching the code directly, as with a static linker. This means that for references from module to module and from module to main program, no indirection is needed, as is the case with most dynamic linkers. This approach however, introduces a problem on some machines, such as RISC machines, where branch offsets do not cover the full address space. For example jumps on the 32-bit ARM architecture have to be within 32 MBytes. This can be solved by introducing indirections in the few cases in which the problem occurs.
For references from the main application to a loaded module the function

```
void *dol_get_symbol (char *name);
```

is provided. This function searches through the symbol tables of loaded modules and returns a pointer to the location of the requested symbol.
Unloading can be achieved by the following 2 functions:

```
int dol_unload_module (char *name);
int dol_unload_module_by_type (int type);
```

These functions take the name or the type of the module respectively. Unloading by type allows the unloading of an OSM without the caller needing to know the name of the current OSM of that type. This is possible because there is only one instance of any OSM type at any one time.

6 Performance

6.1 Overview and Experimental Setup

In order to investigate the characteristics of the code loading system, a series of experiments were carried out.

All experiments were run on an Atmel AT91M40800-based development board (32MHz), with 4MB of external RAM and Cirrus Logic CS8900A 10Mbps ethernet chip. The GCC 3.2.2 compiler and GNU assembler 2.13.2.1 were used to build Arena, the test application and the process managers. The application server and the module server, from which modules were loaded, ran on an Intel PC (Intel Celeron 566MHz, 128Mbyte RAM) running Linux kernel 2.4.19.

The module server was compiled using GCC 2.96. The network link between the development board and PC was a dedicated 10Mbps ethernet link.

Modules of different code size and with different numbers of relocation entries were loaded. The modules were chosen to be real-world code examples, rather than artificially created test modules. Some of the modules were individual relocatable objects files from the Arena C library (libc.a). Other modules used were those from the case studies which have been mentioned in section 1 and are described in detail in [2].

Two measurements were taken for each file:

- The **network transfer time** is the time it takes to request a module over the network and transfer its sections onto the embedded system.
- The **link time** is the time it takes to link the newly loaded module into the application. This involves processing all the relocation entries in the module's relocation table.

These two times added together represent the **total object load time**. The timer resolution was 1ms.

6.2 The Influence of Code Size on Load Times

Figure 5 shows the network transfer and link times of modules with increasing code size. It can be seen that the network transfer time generally increases with the code size. There is one exception at 940 bytes, where the transfer of the modules is faster than the transfer of the next smaller module. A closer inspection of the module has shown that in this particular case the code is contained in a relatively small number of sections compared to the other modules. The ALP protocol transfers the modules by sections, so the transfer of the module with fewer sections is stopped and restarted less often. This might explain the slightly faster network transfer.

Although, in general the network load time increases with code size, as might be expected, the increase in link time does not to seem to follow a regular pattern. Overall there seems to be a trend to increasing link times as the code size is increased, but there several large variations from the expected visible in the diagram. These variations seem to suggest that code size is not the only factor which influences the link time.

6.3 Influence of Relocation Entries on Link Time

To investigate the behaviour of the link time in more detail the number of relocation entries in each module has been counted. Linking, as it is performed by the DOL, involves processing all these relocation entries.

Figure 6 shows the link time for each module. It can be seen that there does not seem to be regular pattern in the increase of link time, as the number of relocation entries increases. There are several surprising sets of results. For example the linking of a module with 60 relocation entries took 25ms, whereas

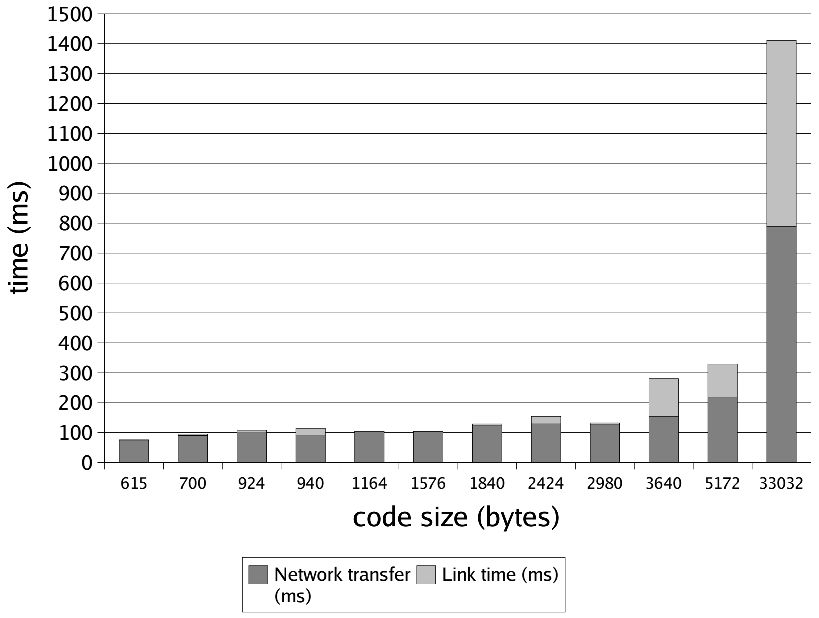

Fig. 5. The influence of Code Size on Network Transfer Time and Link Time

a the linking of a module with 74 relocation entries took 127ms. This increase in link time seems very un-proportional. A module with 86 relocation entries only took 110ms to be linked.

These results seem to suggest that the number of relocation entries has little influence on the link time.

6.4 Influence of Number of Unresolved Symbols on Link Time

In order to further investigate the factors that influence the link time, the relocation entries were looked at in more detail. It was noted that some relocation entries refer to module internal relocations and some to unresolved symbols. Internal relocations do not require the symbol tables of other modules to be searched, whereas finding unresolved symbols introduces further processing.

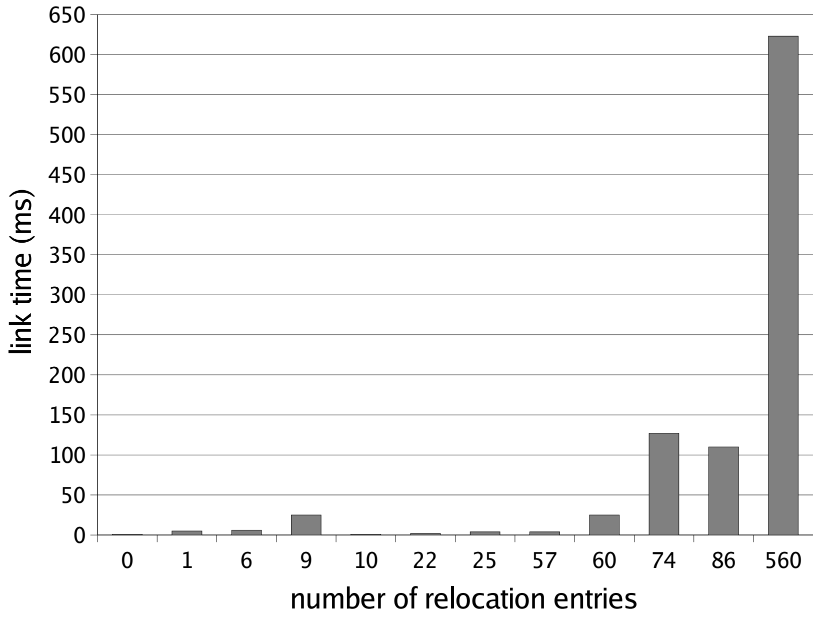

Fig. 6. The Influence of the Number of Relocation Entries on Link Time

Therefore, internal relocations were removed from the analysis and the number of relocations referring to unresolved symbols were counted for each module.

Figure 7 shows the number of unresolved symbols per module together with their link time. Those modules that did not have any unresolved symbols had a link time which was too close to the timer resolution to represent meaningful measures. Therefore, these measurements with zero unresolved symbols are not included in the figure, although these figures lie within the trend of the figure. It is clear from this analysis that the increasing number of unresolved symbols seems to be the major factor in the increase in link time.

There is one surprising result at 37 unresolved symbols. The module seems to load faster than the module with 29 unresolved symbols. Closer inspection of the module has revealed that the majority of the unresolved symbols in that fast-linking module are located in the symbol table of another small dynamically-loaded module, rather than in the main application. Therefore, the size and

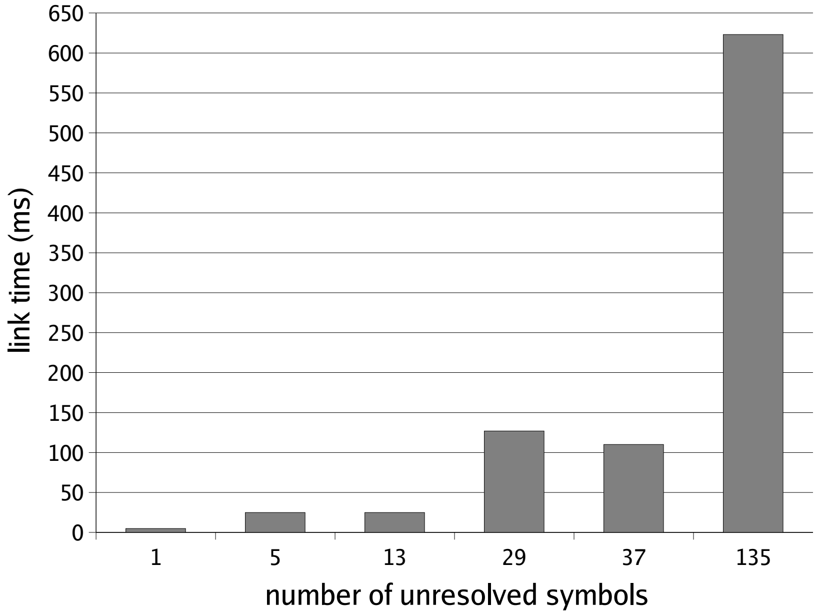

Fig. 7. The influence of the Number of Unresolved Symbols on Link Time

location of the symbol tables that are searched seem to influence the link time as well.

However, an experiment in which the modules were loaded in a different order found no significant variation in the link time. Changing the order in which modules are loaded changes the order of the entries in the `module_info` data-structure. Therefore, when subsequent modules are loaded the symbol tables should be searched in a different order.

6.5 Discussion

The results have shown that the code size of the modules that are loaded are the largest influence on the network transfer time. The largest influence on the link time is the number of unresolved symbols.

In general the network transfer time seems to be greater than link time, but the ratio changes, as the number of unresolved symbols increases. Therefore, the ratio between network transfer time and link time depends on the nature of the application; that is on the interactions between different modules.

Note that most of the modules used for these experiments were extracted from the Arena C library. The Arena C library is a simple C library, which has not been optimised to reduce the number of cross-references between modules. Many libraries are optimised by bundling code into module files in a way that minimises cross-references between modules. This reduces the number of unresolved symbols. Optimising libraries in this way could improve the load performance of the DOL.

7 Conclusion

It has been shown that embedded operating systems can be configured dynamically by loading and linking code into the system at run-time. The Arena operating system provides a platform in which OSMs reside in user-level libraries. A system has been developed which allows the loading of relocatable pieces of code into the running system. This system has been successfully used to replace low-level operating system components, such as the process manager and network protocols.

The dynamic code loading approach can be used in embedded systems that require a high degree of flexibility, but also have to operate with limited resources. The system can be used for operating system configuration, but also as the basis for a flexible and efficient component system.

The results presented in this paper show the load time behaviour of the system. Further experiments [3] have shown, that the run-time overhead, ones the loading of components has been achieved, is minimal.

The system has been used in two case studies to replace process managers and load network protocols dynamically. Furthermore, the use of the system as the lowest layer in a framework for evolvable software systems is currently being investigated in the ArchWare project [27].

References

1. Gingell, R.A., Lee, M., Dang, X.T., Weeks, M.S.: Shared libraries in sunOS. Proceedings of the USENIX 1987 Summer Conference (1987) 131–145
2. Beyer, S.: Dynamic Configuration of Embedded Operating Systems. PhD thesis, University of Manchester (2004)
3. Beyer, S., Mayes, K., Warboys, B.: Dynamic configuration of embedded operating systems. In: WIP Proceedings of the 24th IEEE Real-Time Systems Symposium. (2003) 23–26
4. Postel, J.: Transmission Control Protocol — DARPA Internet Program Protocol Specification – RFC 793. (1981)
5. Berners-Lee, T., Fielding, R., Frystyk, H.: Hypertext Transfer Protocol – HTTP/1.0 - RFC 1945. (1996)

6. Postel, J.: User Datagram Protocol- RFC 768. (1980)
7. Mayes, K., Quick, S., Bridgland, J., Nisbet, A.: Language- and application-oriented resource management for parallel architectures. In: ACM SIGOPS European Workshop. (1994) 172–177
8. Morrison, R., Balasubramaniam, D., Greenwood, M., Kirby, G., Mayes, K., Munro, D., Warboys, B.: A compliant persistent architecture. Software - Practice & Experience, Special Issue on Persistent Object Systems **30** (2000) 363–386
9. Kingsbury, S., Mayes, K., Warboys, B.: Real-time arena: A user-level operating system for co-operating robots. In: Proceedings of The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), CSREA Press (1998) 1844–1850
10. Beyer, S., Mayes, K., Warboys, B.: Application-compliant networking on embedded systems. In: Proceedings of the 5th IEEE International Workshop on Networked Appliances. (2002) 53–58
11. Bovet, D.D., Cesati, M.: Understanding the Linux kernel. O'Reilly (2000)
12. Henderson, B.: Linux loadable kernel module howto (2001)
13. Rashid, R., Baron, R., Forin, A., Golub, D., Jones, M., Orr, D., Sanzi, R.: Mach: A foundation for open systems. In: Proceedings of the Second Workshop on Workstation Operating Systems. (1989) 109–113
14. B. Mukherjee, K.S.: Experimentation with a reconfigurable micro-kernel. In: Proceedings of the USENIX Microkernels and Other Kernel Architecture Symposium. (1993) 45–60
15. Stewart, D.B., Volpe, R.A., Khosla, P.K.: Design of dynamically reconfigurable real-time software using port-based objects. Software Engineering **23** (1997) 759–776
16. Li, Y., Tan, S.M., Sefika, M.L., Campbell, R.H., Liao, W.S.: Dynamic customization in the μchoices operating system. In: Proccedings of Reflection '96. (1996)
17. Bershad, B.N., Chambers, C., Eggers, S.J., Maeda, C., McNamee, D., Pardyak, P., Savage, S., Sirer, E.G.: SPIN - an extensible microkernel for application-specific operating system services. In: ACM SIGOPS European Workshop. (1994) 68–71
18. Pike, R., Presotto, D., Dorward, S., Ritchie, D.M., Trickey, H., Winterbottom, P.: The inferno operating system. Bell Labs Technical Journal **2** (1997)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading, MA (1997)
20. Object Management Group: The common object request broker: Architecture and specification (1995)
21. Sun Microsystems: Jini[tm] Architectural Overview. (1999) http://wwws.sun.com/software/jini/whitepapers/architecture.html.
22. Buck, B., Hollingsworth, J.K.: An API for runtime code patching. The International Journal of High Performance Computing Applications **14** (2000) 317–329
23. Tools Interface Standards - TIS: Executable and Linkable Format (ELF), version 1.2, Portable formats specifications. (1995) http://x86.ddj.com/ftp/manuals/tools/elf.pdf – access date: 29 July 2002.
24. Ho, W.W., Olsson, R.A.: An approach to genuine dynamic linking. Software - Practice and Experience **21** (1991) 375–390
25. Sollinsl, K.: The TFTP Protocol (Revision 2) – RFC 1350. (1992)
26. Postel, J.: Intenet Protocol — DARPA Internet Program Protocol Specification – RFC 791. (1981)
27. The ArchWare research project: Project website http://www.arch-ware.org (2005)