

# Dynamic Loading in an Application Specific Embedded Operating System

Stefan Beyer, Ken Mayes and Brian Warboys

Centre for Novel Computing

Department of Computer Science

University of Manchester

M13 9PL

United Kingdom

Email: {beyer,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 two case-studies are described to show the flexibility of the system.

Performance measurements are presented to show that there is no measurable overhead introduced by the dynamic loading framework after the actual installation of a resource manager. This paper describes the results obtained by the work presented as “work in progress” at the 24th IEEE Real-Time Systems Symposium [1].

## I. INTRODUCTION

Embedded systems are special-purpose systems. They are often designed to perform very specialised tasks. Any operating system running on such a specialised system could benefit greatly from adapting to specific 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.

**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: (1) The system should allow low-level resource managers to be configured to allow maximum flexibility. (2) The run-time overhead should be minimal. (3) The memory footprint should be small. (4) The system should not require a hardware memory management unit. (5) Re-configuration should be reasonably fast compared to the lifetime of a long-lived application. (6) Real-time computing should be possible in between re-configurations.

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 dynamic code-loading approach has been applied to the Arena library operating system, in which Operating System Managers (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 [2], in that the loading and linking of the OSM happens during execution, rather than at application-load-time<sup>1</sup>.

Two possible target applications have been implemented to demonstrate the flexibility of the system. The first of these case studies is a system which allows the replacement of the process manager (PM). The application specifies a new required

<sup>1</sup>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.

scheduling policy and the system loads an appropriate PM over the network and links it into the application. Results of performance experiments are given to show 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. The costs of the loading and linking phases are also given.

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 [3] and HTTP [4] 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. A possible target application could be a multimedia device, which uses UDP [5] to stream data most of the time, but might occasionally be re-configured using remote login over TCP. Using dynamic protocol loading, it would be possible to save memory otherwise occupied by TCP for the actual streaming data.

Section II describes the Arena Operating System. This is followed by a short introduction to the basic concept of dynamic code-loading in Arena (section III) and a discussion of previous work (section IV). Section V describes the underlying code-loading system that was developed for this research and section VI describes the PM loading case study. Section VII gives the results of performance experiments. Section VIII describes the network protocol loading case study. Finally, section IX concludes the paper.

## II. THE ARENA OPERATING SYSTEM

The work described here is based on the Arena library operating system. Arena is an application-oriented operating system [6] [7] intended for both distributed and real-time applications [8] [9]. Operating system policy resides in resource managers implemented as user-level libraries which are linked to the application. The effect of this is to move operating system policy up into the application run-time system. Low-level mechanisms are provided by a hardware-specific nanokernel, the hardware object (HWO). The HWO presents a generic view of low-level processor features. In order to access the low-level mechanisms, resource managers make downcalls to the HWO interface. Conversely, 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. 1 shows how a hardware interrupt may cause the HWO to invoke the user-level PM, which schedules a user-level event handler thread.

## III. DYNAMIC CODE LOADING

OSMs are placed in libraries at user-level in Arena, and it is a logical development to allow the application to load dif-

ferent versions of these libraries dynamically as a means of re-configuring the operating system. The PMs discussed in the present work are implemented as user-level resource managers in libraries. Formerly, these were statically-linked to the application to achieve re-configuration. This work introduces the dynamic loading and replacement of user-level resource managers on Arena and uses a PM case study to demonstrate the flexibility and performance of a dynamic code-loading approach for embedded system configuration. Furthermore, the memory saving advantages of the approach are demonstrated by the network protocol loading case study.

## IV. PREVIOUS WORK

### A. Dynamically Configurable Operating Systems

Many conventional monolithic operating systems allow modules to be loaded into a running kernel. Linux and its kernel module loader [10][11] are a readily available example. Conventional micro-kernel-based systems, such as Mach [12], 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, 4 and 6.

The Kernel Toolkit (KTK) [13] and Chimera [14] 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.

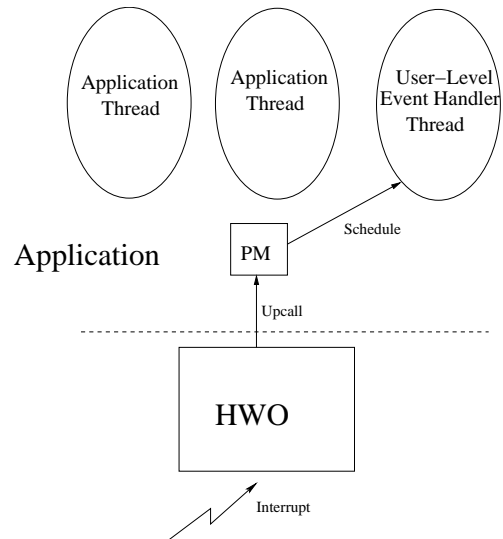


Fig. 1. Arena Event Handling

Systems based on scripting ( $\mu$ Choices [15]), type and pointer-safe kernel extensions (Spin [16]) or virtual machines (Inferno Operating System [17], Java [18]) do not allow configuration of certain low-level resource managers and therefore violate requirement 1, with some of them violating other requirements as well.

### B. Dynamic Code Loading Systems

Distributed systems, such as CORBA [19] or Jini [20] “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 [21] 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 [22] 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 [23] 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 unloading 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).

## V. THE DYNAMIC OBJECT LOADER

A dynamic object loader (DOL) has been developed for the Arena operating system. Figure 2 shows an overview of Arena with the DOL and a process manager switcher (PMS). The PMS is described in the following section. In the Arena HWO nanokernel, 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 [24] and is implemented directly on IP [25] in the prototype implementation. It provides the DOL with a simple send and receive interface, which allows the transfer of modules (MOD in figure 2) 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 3). 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 described below. The application is loaded by the HWO using its application loader component (“Appl.Load” in figure 2). 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.

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<sup>2</sup>. 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

<sup>2</sup>Not all sections of the ELF relocatable file containing the module have to be loaded.

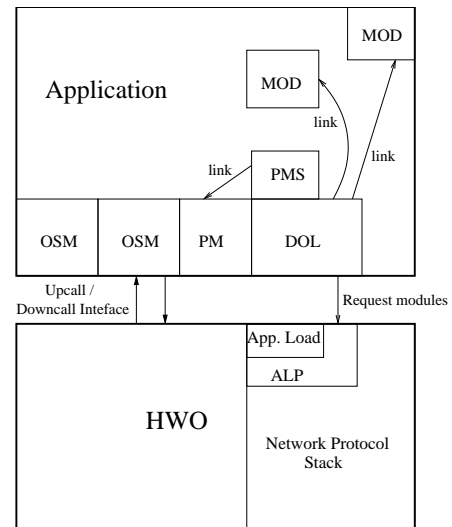


Fig. 2. System Overview

```
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);
```

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.

## VI. REPLACING OPERATING SYSTEM MANAGERS AT RUN-TIME

The loading framework described above can be used to load regular modules, to extend the functionality of a program, or to replace parts of the program with different implementations. For example, sorting algorithms could be replaced in order to use the most efficient algorithm for a certain type of input data.

The DOL could be used to replace OSMs by the application directly, without an additional layer, but this would mean the application programmer having to be aware of the implementation details of OSMs. Typically an OSM includes some state which has to be saved and transferred to the new version of the OSM. An application programmer would need detailed knowledge of internal OSM data structures and the OSM routines that affect the data structures. It is better to provide a simple safe framework for replacing each type of OSM. As a case study the PMS has been implemented. It provides a framework for switching between PMs.

In Arena a PM has to define a certain interface. This interface provides the application with a consistent way of creating, yielding, suspending and switching threads. The PM also allows the application and other OSMs to register event handler threads for hardware events (e.g. interrupts). Application threads and event handler threads are scheduled from `i_switch`, the central scheduling routine. Not every PM will implement the full PM interface, but “null-functions” should be provided for all unimplemented functions.

The PMS provides the following routine to load a first PM at application start time:

```
int load_initial_pm (int type);
```

This call effectively replaces the call to the initialisation routine of the previously statically linked PM. The `type` argument specifies the scheduling policy of the PM to be loaded. For example the call `load_initial_pm (PM.TIME_SLICE)`

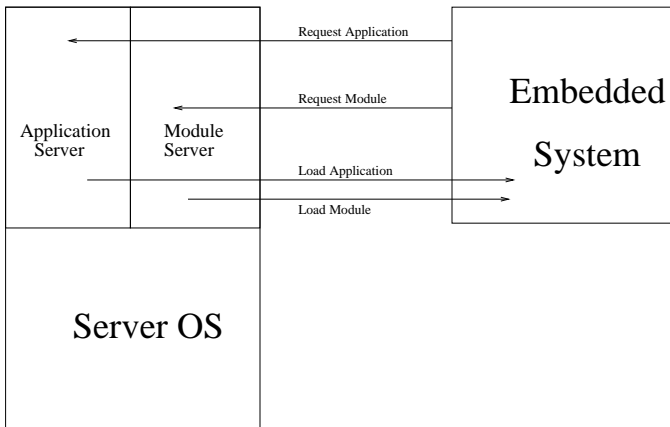


Fig. 3. Application and Module Server

causes the PMS to initialise the DOL, look up the file name of a PM implementing a time-slicing scheduling policy in an internal mapping table, load it using `dol_load_module` and initialise it.

The PM switcher routine

```
void enter_pm_sw (int type);
```

can then be used to replace an existing PM with a new PM of implementation by the application programmer at any time during execution. `enter_pm_sw` causes the currently executing thread to be saved into its data structure. It is necessary to first save the state of the current thread, since the switching of PMs cannot be executed in the context of an application thread. Executing a PM switch using the execution context of a running thread would modify the state of the running thread after saving it, thus leaving the data structures that have to be copied to the new PM in an inconsistent state. Therefore, `enter_pm_sw` uses a special thread context, with its own associated data structure and stack. Arena already provides a separate thread context for the PM's scheduling routine `i_switch`. This special context can be safely used for the switching of PMs. `enter_pm_sw` sets the execution environment to the switch thread context and uses the function

```
int switch_pm (int type);
```

as the entry point for the thread.

`switch_pm` first unregisters all event handlers, so that no event handler can modify the internal PM state during the replacement process. It then saves all the internal PM data structures representing threads. The new PM is loaded using the DOL and the interface of the PM is established using `dol_get_symbol` for each provided function. The PM is initialised with the saved thread data structures. Finally, the switch thread terminates and forces the execution of `i_switch`. `i_switch` then uses the new PM thread state to re-schedule the application threads.

The PMS uses the DOL to link-in PMs directly, by patching code. However in order to do the initial static link of the application, indirections to the external interface of the PM must be used. For this reason PMS provides a collection of pointers to functions, which are assigned to the particular implementations of the functions at load-time by `load_initial_pm`. During the replacement of a PM these indirections have to be updated. The application uses these indirections.

## VII. PERFORMANCE OF PM LOADING

### A. Experimental Setup

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,

	PMRR	PMTS
initial load	263ms	265ms
replacement	279ms	277ms

TABLE I  
PM LOAD TIMES

	PMRR	PMTS
network transfer	200ms	200ms
linking	65ms	62ms

TABLE II  
PM LOADING NETWORK TRANSFER AND LINK TIMES

128Mbyte RAM) running Linux kernel 2.4.19. The module server was compiled using GCC 2.96. For both client and server side, compiler optimisation was disabled. The network link between development board and PC was a 10Mbps ethernet link.

Two implementations of the PM were used for the experiments. PMRR implements a cooperative round-robin and PMTS a time-slicing scheduling policy. PMRR and PMTS were contained in ELF relocatable files of 5172 and 5304 Bytes in size respectively.

### B. PM load time

The purpose of the first experiment was to investigate the overheads of the loading and linking process. It measured the time taken to load a PM dynamically. This included separate measurements for network transport times and link times. A low resolution timer was used (resolution = 1ms).

Table I shows the the total times for the loading and linking process. The first set of measurements are the times taken to load the PM as the initial PM, the second set of measurements represent the time taken to replace the existing PM with the PM specified. Consequently, the "replacement" results in table I contain the time taken to unload the original PM and copy data structures between PMs. As can be seen, this extra processing increased the total time of the loading of PMRR by up to 6%.

Table II shows the results of measurements taken to analyse the actual code-loading process in more detail. The first row shows times taken for the transfer of the PM over the network, and the second row shows the times taken to link the PM into the application. The results show that the network transfer amounts to about 75% of the total load time. A relatively slow network link (10Mbps) was used and the performance would be improved greatly by using a faster network connection.

### C. Overhead of dynamically loaded process managers

The purpose of the second experiment was to measure the overheads of executing the dynamically-loaded PMs. It consisted of comparing the dynamically-loaded versions of the two

		PMRR	PMTS
pm_init	static	240 $\mu$ s	450 $\mu$ s
	dynamic initial load	260 $\mu$ s	460 $\mu$ s
	dynamic replacement	4590 $\mu$ s	4810 $\mu$ s

TABLE III  
PM\_INIT

		PMRR	PMTS
createInitialThread	static	150 $\mu$ s	150 $\mu$ s
	dynamic	150 $\mu$ s	150 $\mu$ s
createThread	static	250 $\mu$ s	250 $\mu$ s
	dynamic	250 $\mu$ s	250 $\mu$ s
yieldThread	static	80 $\mu$ s	80 $\mu$ s
	dynamic	80 $\mu$ s	80 $\mu$ s
m_registerEvHandler2	static	220 $\mu$ s	220 $\mu$ s
	dynamic	210 $\mu$ s	210 $\mu$ s

TABLE IV  
PM ROUTINES

process managers with statically linked versions. The times taken to execute certain PM interface functions were measured and compared. A high resolution timer was used (resolution = 10  $\mu$ s). The following functions were measured:

- pm\_init
- createInitialThread
- createThread
- yieldThread
- m\_registerEvHandler2

The purpose of these functions should be obvious from the name, apart from

m\_registerEvHandler2, which is provided for other resource managers to register event handler threads for hardware events.

Table III shows the durations of pm\_init routine invocations. pm\_init is executed once at the start-up of a PM instance. The “dynamic initial load” and the “dynamic replacement” figures in Table III both represent the time taken for pm\_init of a PM instance which has been dynamically incorporated into an application. “Dynamic initial load” refers to loading a PM into an application which previously had no PM. In contrast, “dynamic replacement” refers to loading a PM into a running application with an existing PM. The order of magnitude difference between the initial and replacement figures represents the cost of copying existing PM state into the new PM instance. This state consists of the data-structures representing threads and a few variables containing information, such as the last scheduled thread. The time taken in pm\_init after dynamic replacement of the PM, although relatively high, is less than 2% of the entire load and link overhead (Table I).

Table III also shows that pm\_init takes up to 220 $\mu$ s longer in PMTS than in PMRR. This difference is due to the extra work that PMTS pm\_init has to do in order to register a timer event handling thread with the HWO. The cost of this extra operation is indicated by the m\_registerEvHandler2 durations given in Table IV. m\_registerEvHandler2 is the PM routine called by other OSMs to establish an event handler thread, which is achieved largely by registering the thread with the HWO.

Table IV gives durations of the other PM routines investigated. The figures for createInitialThread, createThread and yieldThread show that there is no measurable execution-time overhead introduced by the dynamic loading procedure. Figures for each routine are identical for both PM types and for statically and dynamically loaded PMs. The figure obtained for yieldThread duration gives a measure of the cost of the central PM routine, i\_switch. It is difficult to measure the duration of i\_switch directly. This is because i\_switch runs in its own context, and is normally invoked via an upcall from the HWO rather than being called directly. An application call to yieldThread causes the current thread context to be saved, the i\_switch context to be entered, and a new thread selected and scheduled by i\_switch code. yieldThread thus represents the full thread scheduling operation.

There are possibly two pairs of anomalous results given in Tables III and IV. The 4% to 8% difference between the times taken for pm\_init and m\_registerEvHandler2 with static and “dynamic initial load” are not immediately explicable. In one case the static-linked routine is slower than the dynamic-linked routine, whereas in the other case it is faster. This may indicate that the most likely explanation is that the slight differences are due to timing errors.

#### D. Discussion of Results

The results described above show absolute times for the dynamic loading and linking of PMs. It has been shown that the main factor in the total loading and linking time is the network transfer. It is expected that these times could be reduced if a faster network connection was used. The network connection used was at the bottom end of current network speeds. Depending on the target use of the described system a much faster network is likely to be used. Network protocol optimisations might further decrease the total times. There is a penalty associated with saving and restoring the thread data structures between replacement PMs. This approach however is only a small fraction of the total load and link time. The longest total time measured for a re-configuration of the scheduling policy using PMS was 279ms. Any static re-configuration of a system is likely to take much longer. Moreover, it has been shown that once the reconfiguration has been completed, there is no measurable overhead in the execution of a PM, compared to a statically linked version. All other systems described in section IV seem to either introduce overheads which are not acceptable on

embedded systems or lack the flexibility the system described here provides. The structure of Arena, with OSMs placed in user-level libraries, provides the application with access to operating system policies, that cannot be modified in most other operating systems.

## VIII. NETWORK PROTOCOL LOADING

### A. Overview

As a second case study the DOL has been used to implement a network protocol loader. The system allows the dynamic loading of both transport level and application level protocols onto the embedded system. That is, the embedded software can listen to communications ports without the bulk of the associated protocol being loaded. The protocol is loaded when a message needs to be sent or interpreted. Network protocol loading is a use of the DOL which focuses on saving memory as well as on flexibility.

Protocol loading is hidden from the user. That is, some protocol API calls have been replaced by stubs, which load in the rest of the code when required.

### B. Transport Level Protocols

To allow the loading of transport level protocols, the protocol architecture had to be restructured, so that transport level protocols are implemented at user level [9].

It does not make sense to load very small and lightweight protocols dynamically, if the code size of the support code for the loading exceeds the code size of the actual protocol. For this reason only the dynamic loading of TCP and not of UDP has been implemented at the transport layer.

The TCP code has been split into two parts. A static part, containing the code for listening to TCP ports, is linked statically to the application and a dynamic part, containing the main TCP state machine, is located in a module with the module server. The static part provides the equivalent operations to the UNIX system calls *bind* and *listen* and also contains stubs for all the API calls provided by the dynamic part.

There are two cases to distinguish. If the system is used in *server mode*, that is the system waits for incoming connections, the dynamic part is only loaded when a remote system tries to establish a connection to a port the system listens to. *Binding* and *listening* to a port number does not lead to the loading of TCP. If however the system is used in *client mode*, that is the systems tries to establish a connection to a remote system, there is no point in delaying the loading of TCP, since establishing a connection is almost always shortly followed by sending of packets. Therefore a *connect* call causes TCP to be dynamically loaded directly.

### C. Application Level Protocols

The dynamic loading of application level protocols is best achieved by providing stub calls, which load the protocols. This

data size (bytes)	128	256	512
static ( $\mu s$ )	7030	9175	12655
dynamic ( $\mu s$ )	6860	8480	13735

data size (bytes)	1024	2048	4096
static ( $\mu s$ )	18693	32290	60045
dynamic ( $\mu s$ )	19049	31942	60113

TABLE V  
NETWORK DATA “BOUNCE” TIMES

has been demonstrated by implementing an HTTP server. If the embedded application needs to become a web server it calls an initialisation routine, which is a stub call to the network protocol loader. Note that if TCP is not loaded it will be loaded dynamically when a request for a web page is received.

Since all that is normally needed is one stub routine, a huge number of application level protocols can be theoretically supported without using a lot of memory. This could be useful in applications where a large number of protocols are needed, but not necessarily all at the same time. An example of this is grid computing [26]. Embedded systems could participate in a computational grid using such a dynamic network protocol loading system, providing access to computing power on remote machines.

### D. Network Performance

To evaluate the performance of dynamically loaded network protocols, a series of performance experiments have been run. The experiment environment was the same as described in section VII.

Data buffers of various sizes were sent from the PC to the embedded system and “bounced” back to the sending system. The times from the sending of the buffers to the receipt of the bounced buffers were measured using both a statically - and dynamically loaded version of TCP (timer resolution: 1  $\mu s$ ).

Table V shows the average times it took to bounce the data buffers of various sizes. It can be seen that overall the times are very similar in the static and dynamic cases. In some instances the times are slightly lower in the dynamic case, in other instances they are slightly higher. These variations are small and are probably due to variation in network load. Overall the results seem to suggest that there is no measurable overhead in using dynamically loaded TCP.

To establish the cost of loading TCP, the time it takes to establish a connection was also measured. In statically loaded TCP the connection was established at an average time of 17ms. If TCP had to be loaded dynamically during the connecting process it took a total of 1.6s. This suggests that the loading of large protocols, such as TCP, is most effective if the protocol is used rarely. A multimedia device, which uses TCP only for occasional re-configuration would fall into this category.

## IX. 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.

To demonstrate the flexibility and performance of the code loading system, a system which can dynamically load and replace PMs has been implemented. Performance measurements have shown that the loading and linking times are acceptable and most importantly that there is no measurable performance overhead after the replacement of a PM has been achieved.

Furthermore, a network protocol loader has been implemented to show a second use of the system. Rather than focusing on flexibility this case study shows how the system can be used to save memory and therefore allow embedded systems to participate in systems from which they were previously excluded due to lack of embedded resources.

## REFERENCES

- [1] S. Beyer, K. Mayes, and B. Warboys, "Dynamic configuration of embedded operating systems," in *WIP Proceedings of the 24th IEEE Real-Time Systems Symposium*, pp. 23–26, December 2003.
- [2] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared libraries in sunOS," *Proceedings of the USENIX 1987 Summer Conference*, pp. 131–145, 1987.
- [3] J. Postel, *Transmission Control Protocol — DARPA Internet Program Protocol Specification — RFC 793*, Sept. 1981.
- [4] T. Berners-Lee, R. Fielding, and H. Frystyk, *Hypertext Transfer Protocol — HTTP/1.0 - RFC 1945*, May 1996.
- [5] J. Postel, *User Datagram Protocol- RFC 768*, Aug. 1980.
- [6] K. Mayes, S. Quick, J. Bridgland, and A. Nisbet, "Language- and application-oriented resource management for parallel architectures," in *ACM SIGOPS European Workshop*, pp. 172–177, 1994.
- [7] R. Morrison, D. Balasubramaniam, M. Greenwood, G. Kirby, K. Mayes, D. Munro, and B. Warboys, "A compliant persistent architecture," *Software - Practice & Experience, Special Issue on Persistent Object Systems*, vol. 30, no. 4, pp. 363–386, 2000.
- [8] S. Kingsbury, K. Mayes, and B. Warboys, "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)*, pp. 1844–1850, CSREA Press, 1998.
- [9] S. Beyer, K. Mayes, and B. Warboys, "Application-compliant networking on embedded systems," in *Proceedings of the 5th IEEE International Workshop on Networked Appliances*, pp. 53–58, October 2002.
- [10] D. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly, 2000.
- [11] B. Henderson, "Linux loadable kernel module howto," August 2001. <http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/ps/Module-HOWTO.ps.gz>.
- [12] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi, "Mach: A foundation for open systems," in *Proceedings of the Second Workshop on Workstation Operating Systems*, pp. 109–113, 1989.
- [13] K. S. B. Mukherjee, "Experimentation with a reconfigurable microkernel," in *Proceedings of the USENIX Microkernels and Other Kernel Architecture Symposium*, pp. 45–60, 1993.
- [14] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *Software Engineering*, vol. 23, no. 12, pp. 759–776, 1997.
- [15] Y. Li, S. Tan, M. L. Sefika, R. H. Campbell, and W. S. Liao, "Dynamic customization in the  $\mu$ choices operating system," in *Proceedings of Reflection '96*, 1996.
- [16] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer, "SPIN - an extensible microkernel for application-specific operating system services," in *ACM SIGOPS European Workshop*, pp. 68–71, 1994.
- [17] R. Pike, D. Presotto, S. Dorward, D. M. Ritchie, H. Trickey, and P. Winterbottom, "The inferno operating system," *Bell Labs Technical Journal*, vol. 2, Winter 1997.
- [18] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, MA: Addison-Wesley, 1997.
- [19] O. M. Group, "The common object request broker: Architecture and specification. tech. rep. version 2.0," 1995. [http://www.omg.org/technology/documents/formal/corba\\_iiop.htm](http://www.omg.org/technology/documents/formal/corba_iiop.htm).
- [20] Sun Microsystems, *Jini[tm] Architectural Overview*, January 1999. <http://www.sun.com/software/jini/whitepapers/architecture.html>.
- [21] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 317–329, Winter 2000.
- [22] 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>.
- [23] W. W. Ho and R. A. Olsson, "An approach to genuine dynamic linking," *Software - Practice and Experience*, vol. 21, no. 4, pp. 375–390, 1991.
- [24] K. Sollins, *The TFTP Protocol (Revision 2) – RFC 1350*, July 1992.
- [25] J. Postel, *Internet Protocol — DARPA Internet Program Protocol Specification – RFC 791*, Sept. 1981.
- [26] I. Foster, "The anatomy of the Grid: Enabling scalable virtual organizations," *Lecture Notes in Computer Science*, vol. 2150, pp. 1–??, 2001.