

Dynamic Configuration of Embedded Operating Systems

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. 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.

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.

Previously, embedded systems were configured statically at compile time, rather than dynamically at run-time.

Static configuration tends to be more efficient at run-time, but less flexible than dynamic configuration. Static configuration is limited in that the type of specialisation needed may not be known until run-time.

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 sys-

tem 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[1] [2], in which Operating System Mangers (OSMs) are implemented in user-level libraries, linked to the application. Previously, a particular implementation of an OSM, such as a process manager (PM) with a certain scheduling policy, was selected by statically linking to the appropriate PM library.

In the scheme described in this paper, 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 [3], in that the loading and linking of the OSM happens during execution, rather than at application-load-time¹.

As an example of a possible target application of this technique, a system allowing the replacement of the PM, has been implemented. The application specifies a new required 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.

A second target application based on dynamic network protocol loading is currently being implemented.

II. 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 [4][5] are a readily available example. Conventional micro-kernel-based systems, such as Mach [6], place OSMs in user-level servers. An OSM can theoretically be replaced by stopping a server and restarting a different version

¹There are performance improvements with dynamically-linked shared library approaches that delay the resolution of certain symbols until the first reference to them is made at run-time.

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) [7] and Chimera [8] 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 (μ Choices [9]), type and pointer-safe kernel extensions (Spin [10]) or virtual machines (Inferno Operating System [11], Java [12]) 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 [13] or Jini [14] “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 [15] 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 [16] 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 [17] 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).

III. THE DYNAMIC OBJECT LOADER

A dynamic object loader (DOL) has been developed for the Arena operating system. It resides with the OSMs at user-level. The OSMs access low-level mechanisms via a nano-kernel. In this Arena 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 [18] and is implemented directly on IP [19] in the prototype implementation. It provides the DOL with a simple send and receive interface, which allows the transfer of modules 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. 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 loader layer, such as the process manager switcher described below. An application loader, which resides in the nano-kernel, communicates with the application server to load the initial application. Subsequently, modules are loaded by the DOL and the module loader, as required.

The system keeps track of loaded modules in a linked list of data structures, each of which corresponding to one loaded module. This data structure contains the name of the module, the locations and sizes of symbol and string tables and information about all the sections of the module. Not all sections of the ELF relocatable file containing the module have to be loaded. Each module is also given a type. For example, regular modules (i.e. non-OSM modules) are of type REG and process managers of type PM.

The first entry in the module list represents the main application, so that symbol references to the main application can be resolved. Therefore, the first entry is given the type PSEUDO.

Before the DOL can be used it has to be initialised, by an API call, which takes the name of the main application as an argument, so that it can request the symbol and string tables of the application from the module server.

A “load function” is used to request the module’s section header table and section header string table from the server, to load all loadable sections and to “mark” string and symbol tables by pointing to them in the module list entry. 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 containing relocation information and performing each relocation. Undefined references are resolved by patching the code directly, as with a static linker. This means that module to module and module to main program references are direct. A conventional dynamic linker would require indirections for these cases. 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 a “symbol lookup” routine which returns a “symbol handle” is provided.

Unloading can be done by module name, or by module type, allowing a module of a certain type to be removed, without knowing the module’s name.

IV. 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.

The DOL could also be used to replace OSMs by the application. A simple safe framework for replacing each type of OSM can be provided for handling transfer of state between OSM versions. As a case study a process manager switcher (PMS) has been implemented. It provides a framework for switching between PMs.

In Arena a PM has to conform to a defined 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 a central scheduling routine, implemented by the PM.

The PMS provides a routine to load an initial PM, taking the scheduling policy required as an argument. A PM switcher routine can then be used to replace an existing PM with a new implementation. The routine causes the currently executing thread to be saved and moves into a special thread context, with its own associated data structure and stack. First, all Arena event handlers are unregistered, so that no event handler can modify the internal PM state during the replacement process. Next, all the internal PM data structures representing threads are saved. The new PM is loaded using the DOL and the interface of the PM is established using the DOL’s symbol lookup routine for each provided function. The PM is initialised with the saved thread data structures. Finally, the switch thread terminates and forces the execution of the new PM’s scheduling routine.

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, the PMS provides a collection of pointers to functions, which are assigned to the particular implementations of the PM functions at load-time. The application uses these main program to module indirections.

V. PERFORMANCE

A. Experimental Setup

Two performance experiments were run on an Atmel AT91M40800-based development board (32MHz), with 4MB

of external RAM and a Cirrus Logic CS8900A 10Mbps ethernet chip. Two implementations of the PM were used for the experiments. PMRR implements a cooperative round-robin and PMTS a time-slicing scheduling policy.

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 one PM with the PM specified. Consequently, the “replacement” results in table I include 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 relative costs of the network transfer and linking phases of code loading.

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 process managers with the statically linked versions. The times taken to execute certain interface functions were measured and compared. A high resolution timer was used (resolution = 10 μ s).

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.

	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

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 scheduling and dispatching routine.

B. 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 (75 %). It is expected that these times could be reduced if a faster network connection was used. There is a penalty associated with saving and restoring the thread data structures between replacement PMs.

However, this penalty proved to be 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.

VI. CONCLUSION AND FUTURE WORK

It has been shown that embedded operating systems can be configured dynamically by loading and linking code into the

		PMRR	PMTS
pm_init	static	240 μ s	450 μ s
	dynamic initial load	260 μ s	460 μ s
	dynamic replacement	4590 μ s	4810 μ s

TABLE III
PM_INIT

		PMRR	PMTS
createInitialThread	static	150 μ s	150 μ s
	dynamic	150 μ s	150 μ s
createThread	static	250 μ s	250 μ s
	dynamic	250 μ s	250 μ s
yieldThread	static	80 μ s	80 μ s
	dynamic	80 μ s	80 μ s
m_registerEvHandler2	static	220 μ s	220 μ s
	dynamic	210 μ s	210 μ s

TABLE IV
PM ROUTINES

system at run-time. 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.

The replacement of PMs is only one application of the DOL. An investigation into the dynamic loading of network protocols is planned. One possible use of this is to allow embedded devices to participate in Grid computing [20]. It is planned to develop a system in which an embedded system can selectively load and unload protocols according to current application network demands.

REFERENCES

- [1] 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.
- [2] 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.
- [3] 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.
- [4] D. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly, 2000.
- [5] B. Henderson, "Linux loadable kernel module howto," August 2001. <http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/ps/Module-HOWTO.ps.gz>.
- [6] 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.
- [7] K. S. B. Mukherjee, "Experimentation with a reconfigurable microkernel," in *Proceedings of the USENIX Microkernels and Other Kernel Architecture Symposium*, pp. 45–60, 1993.
- [8] 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.
- [9] Y. Li, S. Tan, M. L. Sefika, R. H. Campbell, and W. S. Liao, "Dynamic customization in the μ choices operating system," in *Proceedings of Reflection '96*, 1996.
- [10] 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.
- [11] 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.
- [12] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, MA: Addison-Wesley, 1997.
- [13] 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.iop.htm>.
- [14] Sun Microsystems, *Jini[tm] Architectural Overview*, January 1999. <http://www.sun.com/software/jini/whitepapers/architecture.html>.
- [15] 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.
- [16] 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>.
- [17] 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.
- [18] K. Sollins, *The TFTP Protocol (Revision 2) - RFC 1350*, July 1992.
- [19] J. Postel, *Internet Protocol - DARPA Internet Program Protocol Specification - RFC 791*, Sept. 1981.
- [20] I. Foster, "The anatomy of the Grid: Enabling scalable virtual organizations," *Lecture Notes in Computer Science*, vol. 2150, pp. 1–??, 2001.