FlexRTS: An extensible Orca run-time system†

Leendert van Doorn
Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

FlexRTS is a dynamically configurable and extensible run-time system for Orca, a high performance parallel programming system. It provides run-time and application programmers with full control over the implementation and placement of kernel and user-level modules (device drivers, protocol stacks, thread packages, etc.). This allows programmers to optimize the run-time system on a per application basis and take most leverage out of the available hardware.

Keywords: operating systems, run-time systems, parallel programming, extensibility.

1. Introduction

It is hard for an application programmer to take full advantage of existing hardware. This is largely caused by a lack of control over the available abstractions. Many researchers have interpreted this as kernel abstractions and have proposed mechanisms for extending or adapting these [4, 5, 8, 11, 14]. Non-kernel abstractions, e.g. those provided by a run-time system, are in theory easy to adapt and extend, but in practice they are just as rigid as kernel abstractions. The problem is even worse for micro and smaller kernels where a lot of the traditional operating services are off-loaded to the run-time system adding to its complexity.

In this paper we argue that, in order for an application to take full advantage of its hardware and be most efficient, it should not only have the ability to control and extend its kernel and system servers, but also its run-time system. That is, the programmer should be able to control which particular implementation of an abstraction is used, and when necessary, replace it by one providing extra functionality. Furthermore, the programmer should be able to decide whether a particular implementation is situated in the kernel or user address space. For example, consider a program that uses an unreliable datagram service. Using late binding, the invoker of the program can override the programmer's defaults and specify a different implementation at run-time. This might include using Ethernet hardware as its datagram service when all its clients are known to be on the same segment, and even loading the Ethernet device driver in its own address space when it is not shared among other processes. These kinds of decisions have a major impact on the performance of a program, but are unforeseen at compile-time.

In order to validate the viability of extending run-time systems we are building a new flexible run-time system for Orca [1] on top of the Paramecium [14] operating system. Orca is a distributed and parallel programming language based on the shared object model: a shared memory abstraction that is encapsulated in objects. The Orca system is designed for loosely coupled machines connected by a high-speed network. The latency that is introduced by network communication software is the paramount performance bottleneck in the Orca system. By reducing this we can improve the performance of many course grained parallel applications and in addition run a larger class of fine-grained applications.

Our new run-time system, called FlexRTS, allows the run-time programmer, application programmer, and even the user of the program to specify the implementation of its modules. It even allows modules to be placed securely in the kernel's address space and access kernel services directly. Conversely, traditional kernel services, like device drivers, can be placed in the run-time's address space. This is useful, for example, to reduce copying. All though we are using the Paramecium kernel as our target platform, the techniques used are

†This is a revised version of a paper that has been submitted for publication to the IEEE Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI).

general enough to be applied in other operating systems as well.

In the remainder of this paper we will describe in section 2 Paramecium and its extension model on which FlexRTS is based. Section 3 describes FlexRTS and discusses what components are useful to extend. Section 4 describes an example of efficient shared object invocation in FlexRTS, followed by a related work discussion in section 5.

2. Extensibility in Paramecium

Paramecium [14] is a highly dynamic nano-kernel-like system for building application specific operating systems. Central to its design is a common software architecture for its operating system and application components [7, 12]. Together these form a toolbox. The kernel provides some minimal support to dynamically load a component out of this toolbox either in the kernel or in a user address space and make it available through a name space. Determining which components reside in user and kernel space is established by the user at execution-time.

To support the toolbox of components approach, we use a simple, programming language independent, architecture that provides object instances and interfaces as its main abstractions. In our architecture an object is a collection of methods and instance data. Each object exports one or more named interfaces. This provides support for evolving and generic interfaces. An interface is a set of methods, state pointers, and type information. Objects can be operated on only through the methods in the interfaces they export. Objects are relatively coarse grained.

When inserting object code into the kernel, Paramecium takes the point of view that it is essentially extending the trust relationship. Maintaining this is one of the most important tasks of the kernel. To extend trust, which is a well known security concept, Paramecium uses a certification authority or one of its delegates to sign components it deems trustworthy. These are therefore permitted to run in the kernel its address space. The component signatures are validated at load-time. Depending on a certification authority enables Paramecium to define an informal security model rather than a formally strict one necessary for automated verification. A similar certification mechanism is currently being used for down-loading code over the Internet [10].

Each object has its own instance name and is registered in a hierarchical name space together with one of its interfaces. The hierarchy is reflected in the object's name and is used by other objects to bind to it. Standard operations exist to bind to an existing object, dynamically load one from a repository, and to obtain a different interface from a given object interface. Binding to an object happens at run-time. To reconfigure a particular service you override its name. A search path mechanism exists to control groups of overrides. When an object is owned by a different address space the name service automatically instantiates proxy interfaces.

The Paramecium kernel itself is relatively small, 50KB on a Sun SparcClassic, a 50 MHz machine, and its cross context IPC latency is less than 2 μsec. The kernel defines a small number of other services that we consider essential: memory and context management (physical and virtual), events (synchronous and asynchronous IPC), a secure random number generator, and a device manager. This last service arbitrates between drivers to get access to devices. Everything else is loaded on demand. This extreme position allows us to determine exactly what components we need.

A word of caution is in order. All though the system as we describe it is a general one, we do not anticipate that every application programmer or user will take advantage of its flexibility. It is probably a small set of very special purpose applications (i.e. parallel programs, set-on-top boxes and controller software, high performance web and file servers, routers and gateways, etc.), that will be specially tuned to obtain the performance improvements.

3. An extensible run-time system for Orca

Orca [1] is a programming language based on the shared object model: a shared memory abstraction. In this model the user has the view of sharing an object among parallel processes and invoking methods on it. It is the task of the underlying run-time system to efficiently implement this view. For example, in the current implementation a shared object is either a single copy or fully replicated depending dynamically on the read/write ratio of the object state.

The Orca run-time system is responsible for implementing I/O, threads, marshalling, group communication, message passing, and RPC. With these components it implements the shared object semantics and many optimizations. The current run-time system [2] implements a number of these components and relies on the underlying operating system for others. It is *statically configurable* in that it requires rebuilding and some redesigning at the lowest layers when it is ported to a new platform or when support is added for a new device.

In FlexRTS we enhance the Orca run-time system to take advantage of Paramecium's extensibility mechanisms. The ability to dynamically load components enables us to specify new or enhanced implementations at run-time. Combined with the ability to load these implementations securely into the kernel it is possible to build highly tuned and

application specific operating systems. Important uses of extensibility for FlexRTS are: performance enhancements, debugging, tracing, and controlling individual Orca objects.

An individual object instance is controlled by instantiating it in a programmer defined place in the name space and controlling its search path. Name spaces are defined per process. For example, assume we have a component called ‘‘`/program/shared/minimum`’’, representing a shared integer object. This shared integer implementation requires a datagram service for communicating with other instances of this shared integer object. By associating a search path with the component name, we can control which datagram service, registered under the predefined name ‘‘`datagram`’’, it will us. When no search path is associated with a given name its parent name is used recursively up to the root until a search path is found. This allows us to control groups of components.

The advantage of controlling user level components at binding time is foremost performance improvements, followed by debugging, and a sane failure model. Individual shared object implementations can use different marshalling routines, different network protocols†, different *networks*, debugging versions, etc. On machines where the context switch costs are high, all of the protocol stacks and even drivers for non shared devices can be loaded into the run-time system to improve its performance. In addition this can be used to reduce the copying of packets [3].

Placing components into the kernel is useful for performance improvements and availability. The performance improvements are the result of a reduced number of context switches and the direct access to devices which are shared among other processes. Drivers for these cannot be loaded into user space.

On time-sharing systems it is often useful to place services that are performance bottle-necks in the kernel for availability reasons. These are always runnable and usually do not get paged out. For example, consider a collection of workstations computing on a parallel problem with a job queue. The job queue is a perfect candidate to be down-loaded into the kernel. Requests for new work from a remote process should preferably be dealt with immediately without having to wait for the process owning the job queue to be paged or scheduled in.

Hybrid situations where part of the component is in the kernel and part in user space are also possible. Take, for example, the thread package on our implementation platform. Because of the SPARC architecture each thread switch requires a trap into the kernel to save the current register window set. To amortize this cost we instantiate the thread package scheduler in the kernel, but its synchronization primitives (mutexes, semaphores, condition variables, etc) are instantiated in user and kernel space for fast access.

Although possible, it is undesirable to load the whole program into the kernel. It is important for time-sharing and distributed systems to maintain some basis of trust that, for example, can be used to talk to file servers or reset machines. Adding new components to it should be done sparingly.

4. Example: Efficient shared object invocations

To get some idea of the trade-offs and implementation issues in a flexible run-time system, consider the following shared integer object:

```
OBJECT SPECIFICATION IntObject;
  OPERATION value(): integer;
  OPERATION assign(v: integer);
  OPERATION await(v: integer);
END;

OBJECT IMPLEMENTATION IntObject;
  x: integer;

  OPERATION value(): integer
    BEGIN RETURN x END;

  OPERATION assign(v: integer);
    BEGIN x := v END;

  OPERATION await(v: integer);
    BEGIN GUARD x = v DO OD END;

BEGIN x := 0 END;
```

Each method of this shared object instance can be invoked remotely. For an efficient implementation we use a technique similar to optimistic active messages [13, 15, 16]. When a message arrives the intended object instance is looked up and the method is invoked directly. When the method is about to block it is turned into a regular thread.

To reduce the communication latency and provide higher availability for this shared object instance we map its code read-only into both the kernel and user address spaces. This allows the methods to be invoked directly by kernel and possibly by the user. The latter depends on the placement of the instance state. Under some conditions the user can manipulate the state directly, others require a trap into the kernel. Obviously, mapping an implementation into the kernel requires it to be signed before hand.

In this simple example, mapping the object instance data as read/write in both user and kernel

———————————

†The sequential consistency guarantees of Orca require some cooperation between protocols that are used by other shared object instances.

address space would suffice, but most objects require stricter control. To prevent undesired behavior by the trusted shared object implementation in the kernel we map the object state as either read-only for the user and read-write for the kernel or visa versa; depending on the read/write ratio of its methods. For example, when the local (i.e. user) read ratio is high and the remote write ratio is high, the instance state is mapped read/writable in the kernel and readable in the user address space. This allows fast invocation of the `value` and `assign` methods directly from kernel space (i.e. active messages calls), and the `value` method from user space. In order for the user to invoke `assign` it has to trap to kernel space.

Another example of extending the kernel is that of implementing Orca guards. Guards have the property that they block the current thread until their condition, which depends on the object state, becomes true. In our example, a side effect of receiving an invocation for `assign` is to place the threads blocked on the guard on the run queue after their guard condition evaluated to true. In general the remote invoker tags the invocation with the guard number that is to be re-evaluated.

For our current run-time system we are hand-coding in C++ a set of often used shared object types (shared integers, job queues, barriers, etc). These implementations are verified, signed, and put in an object repository. For the moment all our extensions and adaptations involve the thread and communication system, ie. low level services. These services provide call-back methods for registering handlers. For a really fast and specialized implementation, for example the network driver, one could consider integrating it with the shared object implementation.

To broaden our scope and get experience with extending and adapting other parts of the system we are considering porting the Java [6] virtual machine. Reimplementing Java classes will most probably impact other parts of the system that are currently unexercised.

5. Related work

For the last four of years many research systems have been developed that focus on adapting and extending operating system kernels [4, 5, 8, 11, 14]. Paramecium distincts itself by using a well-proven techniques: objects, dynamic loading, and naming. Together with certification we push these across the user-kernel level boundary.

Our unit of extension is much more course grained than, for example, SPIN's extensions or the Exo-kernel's ASHs. Consequently the mechanisms for implementing extensions are also simpler, while still giving us the extensibility we need.

Furthermore, in our system we allow placement decisions to be made at run-time by the run-time programmer, application programmer and even the user of the program by overriding names. Depending on the usage of a shared object its environment can be changed to control which services it uses. In addition, its implementation can be placed in the kernel or in user address space. Apart from Plan9 [9], which partly uses the same naming ideas, we are not aware of a similar system.

References

1. H. E. Bal, M. F. Kaashoek and A. S. Tanenbaum, Orca: a language for parallel programming on distributed systems, *IEEE Transactions on Software Engineering 18*, 3 (Mar. 1992), 190-205.
2. H. E. Bal, R. A. F. Bhoedjang, R. Hofman, C. Jacobs, K. G. Langendoen, T. Rühl and M. F. Kaashoek, Orca: a Portable User-Level Shared Object System, IR-408, Department of Mathematics and Computer Science, Vrije Universiteit, July 1996.
3. H. E. Bal, R. A. F. Bhoedjang, R. Hofman, C. Jacobs, K. G. Langendoen and K. Verstoep, *Performance of a High-Level Parallel Language on a High-Speed Network*, Journal of Parallel and Distributed Computing, Feb. 1997.
4. B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers and C. Chambers, Extensibility, Safety and Performance in the SPIN Operating System, *Proc. of the 15th Symp. on Operating System Principles, ACM SIGOPS 29*, 5 (Dec. 1995), 267-284.
5. D. Engler, M. F. Kaashoek and J. O. Jr., Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. of the 15th Symp. on Operating System Principles, ACM SIGOPS 29*, 5 (Dec. 1995), 251-266.
6. J. Gosling and H. McGilton, *The Java Language Environment*, Sun Microsystems, May 1995.
7. P. Homburg, L. van Doorn, M. Steen and A. S. Tanenbaum, An Object Model for Flexible Distributed Systems, *Proc. of the 1st ASCI conference*, Heijen, The Netherlands, May 1995.
8. G. C. Necula and P. Lee, Safe Kernel Extensions Without Run-Time Checking, *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, Washington, Oct. 1996, 229-243.
9. R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey and P. Winterbottom, Plan 9 From Bell Labs, *Usenix Computing Systems*, 1995.
10. S. Sclavos, Authentication Practices & Market Adoption of Digital Certificates, *RSA Data Security Conference*, Jan. 1997. (keynote speech).
11. M. I. Seltzer, Y. Endo, C. Small and K. A. Smith, Dealing With Disaster: Surviving Misbehaved Kernel Extensions, *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Seattle, Washington, Oct. 1996, 213-227.
12. M. Steen, P. Homburg, L. van Doorn, A. S. Tanenbaum and W. Jonge, Towards Object-based

Wide Area Distributed Systems, *Proc. of the International Workshop on Object Orientation in Operating Systems*, Lund, Sweden, Aug. 1995.

13. L. van Doorn and A. S. Tanenbaum, Using Active Messages to Support Shared Objects, *Proc. of the 6th SIGOPS European Workshop, ACM SIGOPS*, Wadern, Germany, Sep. 1994, 112-116.

14. L. van Doorn, P. Homburg and A. S. Tanenbaum, Paramecium: An extensible object-based kernel, *Proceedings of the 5th Hot Topics in Operating Systems (HotOS) Workshop,*, Orcas Island, WA, May 1995, 86-89.

15. T. von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, Active Messages: a Mechanism for Integrated Communication and Computation, *Proc. of the 19th International Symp. on Computer Architecture*, Gold Coast, Australia, May 1992, 256-266.

16. D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek and W. E. Weihl, Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation, *Proc. of the 5th ACM SIGPLAN Notices Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.