Customization Lite*

Marc Auslander† Hubertus Franke† Ben Gamsa ‡
Orran Krieger† Michael Stumm‡

† IBM T.J. Watson Research Center
Yorktown Heights, New York
{marc,franke,okrieg}@watson.ibm.com

‡ Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{ben,stumm}@eecg.toronto.edu

Abstract

There has been a great deal of interest in recent years in developing operating systems that can be customized to meet the performance and functionality needs of particular applications while being extensible to support new policies, new implementations and new interfaces. We describe a structuring technique, called building-block composition, that we are employing for this purpose. The customizability this technique provides to untrusted applications is, while large, less than that provided by some other techniques. However, it results in a more extensible and maintainable system, is easier for untrusted applications to use, and provides a better foundation for overall improved performance, particularly for multiprocessors.

1. Introduction

Conventional operating systems provide poor support for applications to customize resource management policies, implementations, and interfaces. Recognizing this, the research community has responded by developing operating systems that are customizable. Examples include SPIN [1], VINO [15], Exokernel [4], Fluke [5], L4 [11], and the Cache Kernel [3]. All of these systems provide for customizability by allowing applications to *extend* operating system functionality with user code, either by having the operating system direct its requests to user code or by downloading the code into the kernel. In these systems, extensions are intended to be used by trusted and untrusted applications alike.

The approach these systems have adopted can be considered an extreme solution to the problem of providing customizability, as it allows any application to extend the operating system. We believe that this degree of customizability, while very flexible, is largely unnecessary, comes at a significant cost in terms of complexity and performance, and is beyond the ability of most application programmers to exploit (in that it takes a systems programmer to write an extension). In this paper, we describe our own lighter-weight form of customization, based on a structuring technique called *building-block composition*. In this approach we separate the notion of extensibility (i.e., adding new code) and customizability (i.e., selecting code). This separation makes it possible to restrict the ability to add extensions to trusted parties only, but allow all applications to customize operating system functionality (albeit in a more limited way than that allowed by the other systems).

Building-block composition is an object-oriented structuring technique, where each virtual resource (i.e. virtual memory region, network connection, file, process, etc.) is implemented by a different set of objects, allowing resource management policies and implementations to be controlled on a per virtual resource basis. We refer to the objects as building blocks, and the overall implementation of a virtual resource as a *building-block composition*. Customizability is achieved by letting the application specify the building-block composition of the virtual resources created on its behalf, and by letting the application dynamically change the compositions at any time. This allows, for example, every open file to have a different pre-fetching policy, every memory region to have a different page size, and every process to have a different exception handling policy.

In specifying building-block compositions, applications

*Appeared in *Proc. 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May, 1997.

choose from a set of building blocks provided by the operating system and (trusted) third party building block providers, and specify how they are to be connected. Application writers need not be system programming experts, since they do not need to write code, but only need to compose a set of predefined modules. Safety is not an issue for the same reason and because the building blocks verify type safety when they are connected. Because building-block compositions are expected to cover the vast majority of customization needs, it is only infrequently necessary to add new building blocks, and such extensions can then be restricted to trusted agents.[1]

Our solution to customizability can be considered to be much more conservative than the techniques chosen by other research groups. In many ways, building-block composition is not much different from other object-oriented techniques [6] and can be viewed as a specific realization of the Framework approach [7]. These OO techniques have already successfully been applied in commercial operating systems; for example the Unix Vnode interface [8] and Streams facility [14]. We build on this previous work, taking advantage of its strengths with respect to maintainability, extending it to all components of the operating system, and, for customizability, providing a powerful mechanism for applications to control the objects used. Overall, we believe that building-block composition is simpler to use than the other more radical techniques, and results in better maintainability and performance because it does not require the system designer to deal with many of the complexity and security problems the other approaches must face.

The *building-block composition* structuring technique was first developed for the Hurricane file system [9, 10], and is now being employed both for the Kitchawan operating system at IBM research and the Tornado operating system [13] at the University of Toronto. This paper is joint work of the Kitchawan and Tornado groups.

We first describe the motivation for employing building-block composition, then describe the technique, and finally compare our technique to other techniques for making operating systems customizable.

2. Motivation

In this section we describe our motivation for having chosen to use building-block composition for customizability instead of the extensibility techniques chosen by other research groups.

[1] This can be viewed to be similar to, say, the installation of new dynamically loadable device drivers in conventional systems.

Customizability

Our goals for customizability are as follows. First, we expect the main motivation will be to improve performance, and hence we require a mechanism with very low overhead. Second, customizations introduced for one application should add no overhead to other applications running concurrently. Third, untrusted applications should be able to customize the virtual resources they use without affecting the security of the system or other applications. Fourth, it should be easy for applications to perform simple customizations (e.g., choosing particular prefetching policies), and hence the mechanism should be simple to use. Finally, the mechanism should be sufficiently powerful to meet the needs of a wide variety of applications, and it should be possible over time to extend the system to be more customizable. We believe building-block composition best meets these goals, as will become evident in the next section.

In our work, we differentiate between customizability and extensibility. Informally, a system is extended when new functionality (i.e., new code) is added, and customized when an application specifies the functionality to be invoked on its behalf. It is one of the fundamental differences between our approach and those employed by others. In our opinion, systems that employ a single mechanism for extensibility and customizability face the following challenging problems. First, because writing system code in the application is just as complex as when written at the system level, creating extensions is beyond the capabilities of most programmers. Second, the same safety mechanism must be used for both trusted and untrusted extensions, with an attendant performance overhead for both. Third, the interface required for extending the system becomes, in effect, part of a larger system API that must continue to be maintained and kept stable for many years if the applications are to remain operational without change. Finally, compared to the APIs of current conventional systems, the richer APIs are even more difficult to test for correctness, in particular to ensure that there are no security holes. It is partly for these reasons that the extensions these systems allow are often restricted.

Maintainability and Extensibility

Another motivation for using building-block composition is that it requires the system to be developed in a highly modular fashion so that it is easier to extend to support new functionality, new applications, and new platforms. The cost of maintaining current systems has grown to such an extent that it is extremely difficult to extend them, and we believe that this is due to the choice previous operating system designers made to sacrifice modularity for performance. A good example of this is the way IBM's AIX operating system integrates the virtual memory manager and native file system [2]. The memory manager knows the structure of

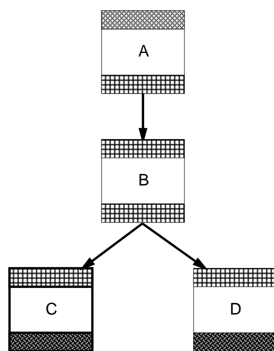

Figure 1. *Building blocks implementing a virtual resource such as a file.* C *and* D *may each store data on a single disk,* B *might be a distribution building block that distributes the file data to* C *and* D*, and* A *might be a compression/decompression building block that decompresses data read from* B *and compresses data being written to* B.

the file system meta data, and traverses this data itself to resolve page faults more efficiently. This optimization makes changing the native file system difficult and makes interactions with other file systems more expensive.

In general, there are a number of reasons why violating modularity leads to long term maintainability and performance problems. First, by doing so, the initial system designer makes the task of future developers, who are often less experienced, more difficult. The interaction between different components of the system may be hard to understand, and hence developers may make incremental changes in a brute force fashion (e.g., cloning code), resulting in poor performance and increased complexity. Second, optimizations typically apply to a particular set of operations considered critical by the designer, and often result in greater complexity and poorer performance for operations which the designer did not consider. Finally, over time, the requirements of the system may change, and the optimizations that were useful initially may make it difficult to adapt the system to the changed requirements. A good example of this is the difficulties many operating systems have had achieving good performance on shared-memory multiprocessors, where optimizations useful for uniprocessor systems (e.g., turning off interrupts to ensure atomicity) resulted in concurrency or complexity problems in multiprocessor systems.

Multiprocessor performance

One of the major differences between our systems and other customizable operating systems is our emphasis on multiprocessor performance. We believe that in the near future single chip shared-memory multiprocessors will become available and soon thereafter the majority of computers will be small-scale multiprocessors. Moreover, we believe that large-scale shared-memory multiprocessors will become increasingly important.

Optimizing for shared-memory multiprocessors requires that the system data be structured to maximize temporal and spatial locality in order to avoid cache conflicts, limit memory and lock contention, and (for large systems) minimize remote memory accesses. To do this, one should avoid global data structures and instead maintain data structures on a per virtual and physical resource basis. For example, the physical pages associated with a memory region should be managed on the basis of that region rather than maintained in a global page cache. This allows the sharing of those data structures (and associated locks) to be constrained to the processors running processes that are actually accessing the corresponding resource, minimizing sharing in the processor caches and avoiding lock contention.[2] With compositions being defined on a per-resource basis, building-block composition directly addresses these issues.

3. Building-block compositions

With building-block composition, each virtual resource instance (e.g., a particular file, open file instance, memory region) is implemented by combining a set of *building blocks*. Each building block implements a particular abstraction and might (1) manage some part of the virtual resource, (2) manage some of the physical resources backing the virtual resource, or (3) manage the flow of control through the building blocks. A building block may be an object (e.g., a C++ object), or might itself be implemented as a composition of other building blocks.

The particular composition of building blocks that implement a virtual resource (i.e., the set of objects and the way they are connected) determines the behavior and performance of the resource. As a simple example, Figure 1 shows four building blocks that might implement some part of a file. B contains references to C and D, and in turn is referenced by A. C and D might each store data on a different disk, B might be a distribution building block that distributes the file data to C and D, and A might be a compression/decompression building block that decompresses data read from B and compresses data being written to B.

It is important to note that *each* virtual resource instance will have a different building-block composition. Thus, two open file instances will be implemented by different sets of building blocks, possibly with different topologies, making it possible to offer highly customized services. An application can (optionally) specify the composition of resources created on its behalf. Also, the composition is dynamic and

[2] On a large system, system data structures should be *de-clustered*; that is, distributed (e.g., replicated, migrated, partitioned) among the different memory modules according to the demands placed on the data [13, 16].

can, in principle, be changed repeatedly by an application (assuming interface requirements are respected).

Customizability

In our building-block framework, customizability can be achieved in a number of ways. First, given a particular composition, it is possible to exchange one building block with another as long as the interfaces of the two are the same. For example, in Figure 1, B could be replaced by another building block B' that implements a different distribution. Thus for each type of building block, multiple implementations may exist, each supporting a different policy or optimized for a different application behavior. Even with only a few implementations of building blocks, the combinatorial effect on the behavior of an entire composition can be huge.[3]

Third, building blocks can be added to an existing structure if the connecting interfaces match, thus modifying the topology. This can be used to add new functionality to the composition. For example, a new building block E (that, say, implements prefetching of some sort) can be inserted between A and B, as long as both the imported and the exported interface of E is the same as that exported by B. Building blocks that import the same interface they export can be arbitrarily stacked. As another example, to implement a replicated file, one can imagine just adding a replication building block F between A and B that is connected to both B and a second subtree rooted by another distribution object similar to B.

Finally, it is possible to present different interfaces to applications by employing building blocks that export these interfaces, but import standard interfaces so that they can be connected to existing structures.

Operation and structure

When an application is instantiating a building-block composition, it instantiates each building block and specifies to the building-block constructor the other objects that building block should interact with. Once the entire composition is complete, the building blocks verify that each referenced object is of the correct type and that any other required constraints are met. Hence, if some building block requires, say, that a particular file block size be supported, it verifies that all building blocks it is connected to can in fact support that block size. This type of checking makes it safe for untrusted users to customize the building-block compositions.

For maintainability (and to make validation feasible), it is necessary to keep the number of building-block interfaces small. This is done by the system designer imposing a structure that defines the types of building blocks that can be in a composition. For example, there are four types of building blocks in the Tornado memory manager that: (1) control the TLB entries specific to a region of memory, (2) provide the mapping between a range of virtual addresses and a file, (3) control the caching of file blocks in main memory, and (4) for each file provide an interface to the corresponding file system. These four define the (only) four basic building block interfaces in the memory manager.

The building blocks that make up one of the basic compositions may each be an object, or a composition of other building blocks. For example, the memory management building block that controls file block caching can be implemented as a composition of one building block for managing the cache with another building block for prefetching file blocks. In this case, both building blocks support the same interface and are composed in a standardized fashion that allows for validation of the composition. (The details are beyond the scope of this paper.)

In some cases, such as the building blocks that control the disk layout of a file, it does not make much sense to change the composition after instantiation. However, generally building blocks do support modifying a composition at run time. For example, an application may want to replace an existing prefetching building block of an open file instance with one that implements a different policy in order to adjust to the access pattern expected in the upcoming phase of the computation. To do this, the client instantiates a new prefetching building block, asks the existing block to transfer the state of any physical resources and connections to the new block, and to deallocate itself. In the specific case of the prefetching building block, there is little state that needs to be transferred. However, in the case of a memory management building block that manages the cache of a file's blocks, this state can be substantial.

Performance considerations

The modularity of building-block composition results in some overhead. However, with a proper design this overhead is typically small for a number of reasons. First, even if a resource is implemented with many layers of objects, requests are often serviced by traversing only a small number of layers. For example, in the Hurricane file system [10], objects typically cache data, and most read and write requests can be satisfied from a cache managed by an object close to the client. Second, building-block composition naturally minimizes the number of cross address space object invocations, because the customization occurs in the server (or kernel) providing the service. In contrast, some of the other mechanisms for customization require the customizations to be implemented in the application space or a separate user-level server. Third, because building blocks are trusted, no checking for security is necessary, and hence

[3] For an extra degree of customizability, many of our building blocks are also parameterizable (i.e. prefetch distance, page size, etc.).

object-invocation overhead is low. Finally, since the functions and data of a building block are specific to the (typically) simple policies it implements, it is feasible to choose an implementation that has less (code and data) overhead than a general purpose implementation.

If the use of many building blocks happens to result in poor performance for some important new workload, then it is feasible for a system programmer to define and add a new object class specifically to handle the demands of this workload, possibly by combining a set of simple building blocks into a single more complex one.

For multiprocessors, the modularity imposed by building-block composition, rather than entailing overhead, is actually required in order to achieve good performance. That is, one can only achieve locality in accessing the state of a virtual resource if that state is encapsulated by objects specific to that resource. In previous work, we described a particular technique for implementing building blocks that exploits the modularity to achieve good multiprocessor performance [13]. It provides a structured (and somewhat automated) way to design building blocks that match any locality in application requests, enabling optimizations for locking and cache/memory locality.[4] For example, the state of the building blocks that manage the physical memory of an application that spans many processors might be distributed across all of those processors, while the state of the building blocks that manage the physical memory of a sequential application might migrate with that application to the processor on which it is executing. We believe that the performance advantages that accrue from optimizing for locality will more than make up for any overhead entailed by an object-oriented design.

4. Comparison to other approaches

Recent examples of other approaches for developing customizable systems include those taken by the SPIN [1, 12], VINO [15], Exokernel [4], Fluke [5], L4 [11], and Cache Kernel [3] projects. The SPIN and VINO operating systems allow applications to extend and customize the operating system by allowing them to down-load untrusted code into the kernel. Distinguishing features of SPIN include the compiler based techniques used to ensure that down-loaded code is safe and the event mechanism used to dispatch down-loaded code. A very interesting feature of VINO is the transactional model used to guarantee error recovery. While these two systems are very different, from our perspective they both (to one degree or another) suffer from performance overheads resulting from the security mechanisms as well as maintainability difficulties which result from exposing internal interfaces that become part of the API. Also, SPIN restricts extensibility by limiting the internal interfaces available to extensions.

The Exokernel, Fluke, L4 and Cache Kernel all allow for customizability by having the kernel redirect hardware events to external address spaces where they can be customized on a per-application basis. An Exokernel is a small kernel that directs hardware-specific events to untrusted user-level libraries that implement all policies. Fluke, L4 and Cache Kernel employ a virtual machine model, where each application can have a different operating system running in a virtualized version of the hardware. To one degree or another, these approaches suffer from the overhead required to cross address space boundaries and they make the sharing of resources between applications desiring different extensions difficult.

The most fundamental difference between our approach and these others is that we have chosen to use separate mechanisms for customizability and extensibility. Customizability is provided by allowing untrusted applications to compose already existing building blocks in new ways. Extensibility is provided by a highly modular object-oriented design that allows system and third party developers to add new functionality to the system. This distinction has the following impact on our system. First, it leads to lower overhead. Since building-block composition is applied on a per-resource basis, an application does not affect the overhead of applications using other resources when it customizes its own resources. In fact, it can be argued that there is no overhead for either customizability or extensibility in our system. Any overhead is intrinsic to the modularity of the system, and this modularity pays for itself by increased maintainability and by improved multiprocessor performance. Second, the interface available for customizability (i.e., specifying and changing compositions) is simple and hence easily maintained. The internal interfaces (between building-blocks) are not exposed to the applications, and hence are not part of the API that needs to remain stable.[5] Third, our system does not need to deal with many of the security issues the other systems face. Finally, the customizability we provide is easy to use — applications compose existing functionality to adapt the system to their needs rather than writing new code.

On the surface, the customizability provided by our approach appears to be more restrictive than that provided by the other customizable systems. While this is indeed the case for untrusted applications, for trusted agents, our system is actually less restrictive than the other systems. All

[4] With this technique, called *clustered objects*, a reference to an object might actually refer to just one component of a larger distributed object. That is, a single object reference can point to different objects on different processors, thus efficiently supporting replication, migration, and distribution of objects behind a uniform object-oriented interface.

[5] Of course, stability of the internal interfaces is still important to the development community that is exposed to these interfaces, but it is less critical than the stability of the general application API.

internal interfaces are available to new code in our system, while the other systems often restrict the interfaces for safety reasons. Also, it should be noted that we are willing to trust code provided to us by authenticated (trusted) applications such as a data base, and allow it to be dynamically linked into the system. Finally, while it is not central to our work, building-block composition can be used as a framework for some of the other techniques being developed for customization. For example, a trusted building block can be provided that accepts and restricts the access of downloaded code from an untrusted application, and another one might be capable of reflecting events into the application address space.

5. Concluding remarks

It is feasible to provide for operating system customizability along a spectrum of possibilities. At one extreme, conventional operating systems offer essentially only minimal customizability. Recently, a number of systems have been designed that lie at the other extreme in that they allow any application to extend and override operating system functionality. In this paper, we presented an alternative solution that lies between the above two extremes; it allows any application to specify how virtual resources are to be implemented by choosing from a set of existing building blocks and connecting them together, but restricts the addition of new building blocks to trusted parties.

Hence, our solution restricts the degree to which any application can customize the system, but we believe that in return, it results in a more efficient system, one that is easier to customize for average application programmers, one that is easier to maintain, and one in which it is much easier to ensure safety. An important question that remains open is how much customizability is really required in the common case, and at what cost. Years of experience running production systems will be required to fully understand many of the tradeoffs involved.

Building-block composition was first developed for the Hurricane file system, and has proven to be successful in the context of the requirements of a file system. However many specific issues have to be re-examined in the more challenging context of a total operating system. Kitchawan is in a very early stage in its design, and Tornado is in a relatively early implementation stage (although it is currently operational) and hence there are many issues still to be resolved. Specific mechanisms that require more work are those for validating that compositions are valid and those for enabling applications to efficiently specify compositions to the system. We will have to have substantially more experience with a full operating system implementation before we have validated our claims of low overhead, extensibility, maintainability, and good multiprocessor performance.

References

[1] B. Bershad, S. Savage, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th Symp. on Operating Systems Principles*, pages 267–284, 1995.

[2] A. Chang, M. Mergen, R. Rader, J. Roberts, and S. Porter. Evolution of storage facilities in AIX Vers. 3 for RISC System/6000 processors. *IBM J. of Research and Development*, 34:105–110, 1990.

[3] K. Duda and D. Cheriton. A caching model of operating system kernel functionality. In *Proc. 1st Symp. on Operating Systems Design and Implementation*, pages 179–193, 1994.

[4] D. Engler, F. Kaashoek, and J. O. Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th Symp. on Operating Systems Principles*, pages 251–267, 1995.

[5] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, pages 137–152, 1996.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[7] N. Islam. *Distributed Objects: Methodologies for Customizing Systems Software*. IEEE-CS Press, 1996.

[8] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. USENIX Conference*, pages 238–247, 1986.

[9] O. Krieger. *HFS: A flexible file system for shared memory multiprocessors*. PhD thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 1994.

[10] O. Krieger and M. Stumm. HFS: A performance-oriented flexible file system based on building-block compositions. In *Proc. 4th Workshop on Input/Output in Parallel and Distributed Systems*, pages 95–108, 1996.

[11] J. Liedtke. On micro-kernel construction. In *Proc. 15th ACM Symp. on Operating System Principles*, pages 237–250, 1995.

[12] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, pages 201–212, 1996.

[13] E. Parsons, B. Gamsa, O. Krieger, and M. Stumm. (De-)clustering objects for multiprocessor system software. In *Proc. 4th Intl. Workshop on Object Orientation in Operating Systems 95 (IWOOOS'95)*, pages 72–81, 1995.

[14] D. Ritchie. A stream intput-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[15] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, pages 213–228, 1996.

[16] R. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9(1/2):105–134, 1995.