

Dynamic Customization in the μ *Choices* Operating System

Yongcheng Li See-Mong Tan Mohlalefi L. Sefika
Roy H. Campbell Willy S. Liao

Department of Computer Science
University of Illinois at Urbana-Champaign
Digital Computer Laboratory
1304 W. Springfield
Urbana, IL 61801
{*ycli, stan, sefika, roy, liao*}@cs.uiuc.edu

Abstract

The lifetime of an operating system is long compared with that of its many varied applications and uses. General purpose or flexible systems design are solutions that address such lifetime differences. General purpose systems suffer from size and inefficiency. Flexible operating systems suffer from the overhead introduced by the mechanisms permitting flexibility. Structured dynamic customization facilities are required in order for the system to remain manageable and memory efficient. However, if such overheads can be ameliorated by performance improvements obtained by tuning the system to application behavior, flexibility has little cost. In this paper, we describe a meta-level architecture for dynamically modifying the resource management policies of a micro-kernel operating system. The architecture supports reflective computation through a flexible scripting language embedded in the micro-kernel. Kernel interpreters ensure a safe environment for script execution, allowing user processes to extend and customize the system's capabilities in non-conventional ways. Since scripts are interpreted, the kernel's safety depends only on the interpretive environment. In such an environment, the running system may generate and add on-the-fly resource management policies without recompiling either the user application or the operating system. Again on-the-fly, the system may incrementally specialize scripts if they are object-oriented. We describe in detail our experiment with implementing a dynamically reconfigurable process management system in the μ *Choices* operating system.

1 Introduction

Dynamic operating system customization has become crucial in order to bridge the ever growing mismatch between application needs and operating system policies[1, 29, 3]. For example, while most operating systems implement some form of the *Least Recently Used* (LRU) policy for page replacement in their virtual memory systems, Stonebraker has shown that LRU within database systems results in suboptimal performance because database systems often access large data sets in a decidedly non-LRU fashion[29]. Therefore, database applications should dynamically tailor the operating system and integrate a suitable page replacement strategy for its pages. In general, no generic operating system policy can ever satisfy all applications; it is important to permit individual applications to customize dynamically the system by specifying or providing specialized policies[12, 2, 15, 14, 33].

Existing operating systems support dynamic customization and evolution in a rather inadequate and limited manner. Both operating system adaptations and application extensions are restricted to compile time modifications. For example, in systems like Mach[23], V++[5], SPIN[3],

and Apertos[32], application-specific meta-level policies are implemented by user-provided *compiled* code. The Synthesis[22] kernel and its offspring, Synthetix[21], adopt a less conventional customization strategy that incorporates reflective computation for run-time code generation. However, the introspection and feedback scheme that triggers the generation of new code is still fixed and pre-determined at compile time. In short, none of the current operating systems provides a *fully open* meta-level architecture for dynamic reconfiguration and customization, where new application-specific policies can be generated on-the-fly, activated, deactivated, or completely removed without excessive recompilation and restructuring.

Our research concerns the construction of an open architecture for reflective computation and dynamic customization using a flexible scripting language in an operating system kernel. As the language is interpreted, the safety of the scheme depends solely on the interpreter. We do not require a trusted compiler in order to embed securely per-application code in the kernel. Applications do not always have to rely on the availability of compilable code in order to gain the benefits of run-time adaptability. The degree of trust afforded to a particular script may dictate the restrictions imposed on the script's interpretation. Scripts may be manufactured on-the-fly or customized by specializing existing scripts. In addition, our technique also solves the notorious problem of monotonically growing operating system extensions and "code-bloat". Scripts are easier to remove from a running operating system than compiled and linked code. The script interpreter itself can be deleted and new instances created as needed. Thus, the μ *Choices* operating system adopts a flexible dynamic customization scheme where user-level policies and mechanisms can be integrated with the system through controlled addition and deletion of scripts, without unnecessary recompilation of parts of the operating system or applications.

In section 2, we describe the *Choices* operating system and the lessons learned from its implementation that motivated dynamic customization (section 3) in its successor, μ *Choices*. We also discuss the motivation for scripting in our reflective architecture, and the role of object-orientation in scripting. We describe a meta-level architecture for dynamically modifying the resource management behavior in μ *Choices* in section 4. A dynamically reconfigurable process management subsystem was implemented in μ *Choices* and this is discussed in section 5. We conclude in section 6.

2 Experience with dynamic OS customization

Studies of adaptive behavior in the *Choices* object-oriented operating system cover a variety of contexts, ranging from message passing[11] to file caching[15] to API extensions[26] to run-time controllable debugging[16]. Islam[11] shows that customized message passing protocols improve the performance of message passing parallel computation over standard PVM implementations. By examining application message passing patterns, customized protocols eliminate unnecessary acknowledgements and bound buffer requirements. Lim[15] demonstrates that adaptive file caching based on changing patterns of file accesses consistently matches or outperforms non-adaptive file caching.

Madany[16] implements a run-time extension facility based on reified classes and class hierarchies in order to adapt activities such as debugging and object persistence. These meta-facilities also support dynamically extensible visualization and browsing in *Choices*[27]. The visualizer allows interactive run-time customization of memory management policies in the system, significantly aiding performance debugging.

Our meta-facilities for run-time customization in *Choices* enable many important operating system features and debugging capabilities. However, in all these, we rely solely on compiled code. This imposes many limitations and consequences, including long edit-compile-execute cycles, slow prototyping of services, and rigid, compile-time decisions, especially in situations when the dynamic behavior is highly unpredictable.

3 Motivation for script-based customization

μ Choices[4, 31] is a redesign of the *Choices* operating system as a micro-kernel operating system. It draws many of its design ideas from the original *Choices*, especially with regard to frameworks that support incremental specialization.

3.1 Interpreted Agents

Interpreted agents within the μ Choices kernel can eliminate the overhead of cross-domain user-to-kernel system call invocations for operating system services when a scripted agent can amortize system call overhead. Scripted agents can remove control traffic between the user and kernel. Trusted scripts can provide kernel level processing of multimedia streams between different transport, storage, and display devices. For example, it is usually unnecessary to pass a video frame coming over the network to application space before displaying it. Instead, a script may control system processing of video frames arriving from network subsystem to display subsystem without interference or crossing protection domains.

In [28], Small and Seltzer measures the efficiency of several different operating system extension languages. They conclude that compiled object code is more efficient than interpreted code as an operating system extension device as the overhead for interpretation is prohibitively large in the two sample scenarios they consider (virtual memory page eviction and compute intensive stream processing). Since compiled code runs much faster than interpreted code, their conclusions have intuitive appeal. However, as Small and Seltzer also point out, the recent introduction of the Java interpreted programming language offers the potential for efficient execution at speeds comparable to compiled code[9]. With current compiler technology, this is entirely plausible. In the current release of Java (beta release 1.0b1), the language compiles to a machine-independent byte-code. Future releases will incorporate compilation from byte-code to native machine code as programs are loaded into the interpreter (“just-in-time compilation”), thus trading greater cost at start up to more efficient later execution. Using scripts at a meta-level to specify and modify policies also helps to eliminate some of the need for high-performance script execution. Based on these trends, we decided to prototype an architecture for reflective computation using interpreted scripts.

3.2 Scripts versus Compiled Code

Interpretive environments offer several advantages over compiled machine code. Source code is required for the manufacture of trusted compiled code by a trusted compiler. The trusted compiler cryptographically signs the object code. The kernel is able to verify the signature through the possession of a cryptographic key shared between the compiler and the kernel. Without the source code, trusted object code cannot be produced. Source code may not always be available; third party software providers often are unwilling to provide source. This problem may be acute — software vendors nowadays routinely distribute kernel loadable modules in object code format (“.o’s”). In order for the kernel to trust compiled code from an external source, it must also possess a cryptographic key it shares with the source, leading to problems in key distribution and management within the kernel. Scripts suffer none of these limitations. No trusted compiler is required, nor is source code needed.

Scripts also solve the problem of “code-bloat.” As the operating system runs, applications load extensions into the kernel in order to customize the system. Compiled code is dynamically linked with the kernel through the use of a dynamic linker. Old and obsolete extensions must be removed (garbage collected), otherwise the memory occupied by the kernel will grow without bound as time goes on. Scripts are more easily discarded than compiled code. The interpreter itself may be deleted, along with the constituent script. A new instance of the interpreter could then be used to run new scripts.

3.3 Object-Oriented Scripting Languages

The Java language[9], in particular, seems well suited for the task of dynamic customization and application-specific customization in an operating system.¹ Designed as a downloadable Internet scripting language, it already possesses the important attribute of *pointer safety*.² As an object-oriented language, it complements the object-oriented nature of the base μ Choices system. Just as objects in the base system’s framework are customizable through inheritance and dynamic binding in C++, scripts in an object-oriented language can be *incrementally specialized* through the same language level mechanisms. This means that it is possible to extend dynamically already running scripts that implement objects in a class hierarchy. Extensions implemented in an object-oriented language bring the advantages of encapsulation, incremental specialization and code reuse to the extension modules themselves.

Tcl[19] is used in the prototype reported here as it was already available as an agent processing language. As we used Tcl in our prototype, it is not possible to explore fully the issues raised here.³ Our preliminary experiences with Java are very encouraging and we intend to report our results in a succeeding paper.

4 Meta-Level Architecture

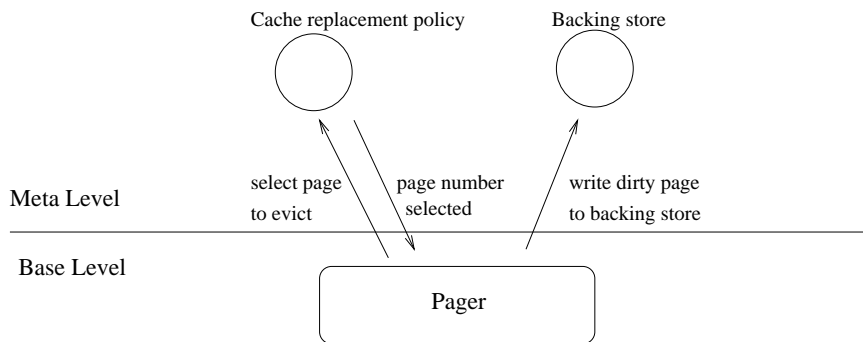


Figure 1: Base and Meta-levels in Virtual Memory Systems.

Meta-object interfaces and protocols are a well-established customization technique in object oriented operating systems[32, 8]. In this approach, the semantics of an object’s behavior is defined by a set of objects, named meta-objects, in its meta-space[32]. Changing or activating different components in the meta-space of an object allows customization of the behavior of the object. Meta-interfaces and protocols help program such complex object associations and interactions in a systematic fashion.

Figure 1 illustrates these concepts in the context of a reflective demand paging system. The base-level *Pager* object manages page faults in terms of updating a cache of pages in main memory. The base level defines primitive virtual memory functionality and hardware management. The meta-level defines the meaning of the operations that are included as part of the pager’s behavior. The meta-level can impose customizable policies (eg., *LRU*, or *Most Recently Used (MRU)*, or *Working Set (WS)*) by binding operations like “find a page to replace” to specific algorithms. The binding may depend on the parameters of an operation. For example, in our system the meta-objects for the pager define the semantics of *page eviction* for each virtual memory region. The *Pager* may be associated with other meta-objects, such as a *backing store*,

¹We discuss other avenues opened up by scripts in section 6.

²Java scripts cannot manufacture pointers to arbitrary locations in memory.

³Packages such as [incr tcl][17] bring object-orientation to Tcl scripts. However, we felt that pursuing further development beyond a “proof-of-concept” point with Tcl to be without merit.

which defines where the data is written to when a page is evicted. Through the meta-interface, the association between the pager and its various policies can be changed dynamically as needed. A particular policy may alter the bindings of several of the pager operations in order that the behavior of the pager be faithfully implemented using the specific policy semantics. For example, a working set policy needs to maintain a window of pages that have been referenced and a page replacement algorithm based on that window. This particular separation of base/meta functionality has been cogently argued in much existing literature[14, 33, 18].

$\mu Choices$ extends this basic meta-hierarchy to two levels. An instance of the hierarchy for a

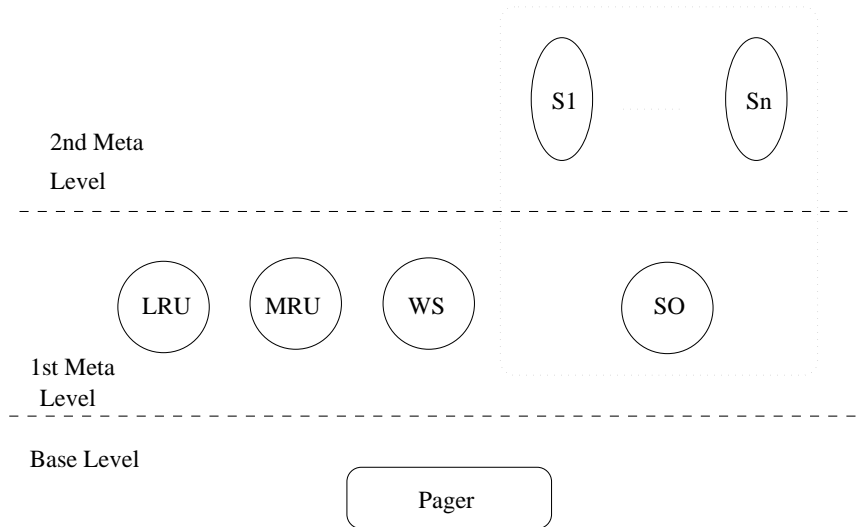


Figure 2: Two level meta-hierarchy in $\mu Choices$ w.r.t. cache replacement policy.

virtual memory system is depicted in figure 2. The first meta-level is similar to that of figure 1. The first level meta-interface is concerned with programming the policy interface of objects such as *Pagers*. Through the first level meta-interface, the page eviction policy, as well as other meta-objects, may be bound to the *Pager* object.

In the $\mu Choices$ meta-level architecture, the meta-objects at the first meta-level include special *scripting objects (SO)*. *The semantics of a SO depend on the script it executes.* Thus the second meta-level is populated by scripts (objects S_1, \dots, S_n in figure 2). Since a scripting object can run *different* scripts, we place scripts at the second meta-level, where they define the semantics of the first meta-level scripting object. The second level meta-interface is concerned with programming the interpretation of the scripts themselves. The meta-operations at the second meta-level include:

- adding scripts,
- deleting scripts,
- updating scripts, and
- running scripts.

Through this second level meta-interface, scripts may be inserted and run to allow application-specific customizations of operating system mappings. The distinction between the first and second meta-levels separates the issues of script control and execution from the problems of controlling policies.

Scripting objects are implemented as in-kernel interpreters. $\mu Choices$ currently includes both the Tcl and Java interpreters. In the next section, we describe the construction of a dynamically customizable process management subsystem based on the meta-level framework discussed here.

5 Process Management

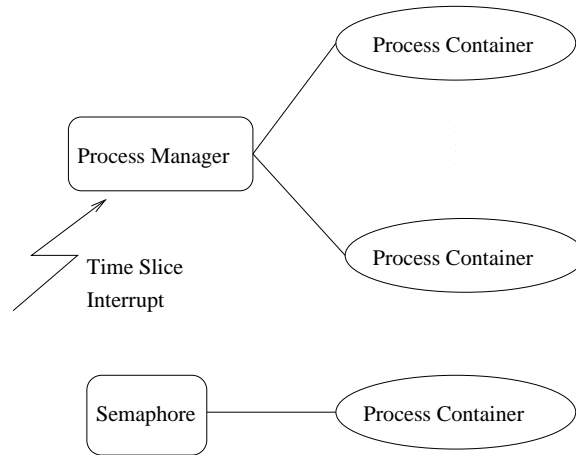


Figure 3: Architecture of the Process Subsystem.

Process management is an integral part of any operating system. The process subsystem is in charge of both system-level and application-level threads. It manages scheduling and context switching between threads, ensuring that the correct virtual memory domain is restored on a context switch. It also handles synchronization between threads through mechanisms such as semaphores. Together with the hardware support and virtual memory management subsystems, the process subsystem forms the core of modern microkernel operating systems.

A flexible process management and scheduling framework would permit the implementation of a large number of scheduling policies and mechanisms that can be responsive to changes in the computing environment. The UNIX operating system, for example, uses hierarchical round robin process scheduling[24]. A fixed policy is inappropriate in many parallel applications since time slicing often causes parallel applications to perform poorly[1]. In addition, gang or co-scheduling, where a set of processes is scheduled together, can significantly improve the performance of tightly coupled parallel applications[13, 20, 7]. In many cases, application control over thread scheduling increases parallelism[2]. The use of *scripting schedulers* permits the dynamic customization of the process management system at run time with scheduling policies not previously envisioned by the system designer.

Figure 3 depicts the architecture of the process management system in μ Choices. Processes are modeled as objects of class *Process*. Instances of the *Process* class contain all the information regarding the execution context of a thread, including processor and virtual memory context. Instances of the *ProcessContainer* class are repositories of *Process* objects. Operations on class *ProcessContainer* allow the addition and removal of *Process* objects from a container. Subclasses of *ProcessContainer* implement different policies regarding addition and removal, such as FIFO or priority-based. *Semaphores* handle synchronization between threads. Each semaphore holds a *ProcessContainer* of waiting threads. Threads that obtain a semaphore lock transit from the semaphore's container to one of the "ready-to-run" containers in the Process Manager.

There is a single instance of class *ProcessManager* residing within the micro-kernel. The process manager is in charge of multiplexing threads among the physical CPUs. In order to do that, the process manager registers for notification of time slice interrupts from the hardware support layer of the micro-kernel. The process manager maintains some number of process containers, which in turn, contain processes that are ready to run. These containers are called "ready-containers." Processes which have similar properties share the same container. A process management system implementing workahead scheduling[10] would, for example, include containers for the three different types of processes: real time processes, interactive processes

and non-real time processes. An application may also construct its own policies by specifying a *ProcessContainer* object be created to group its processes together. Ready-container association is made at run time when processes are created. Threads from different virtual memory domains can share the same container.

When a time slice interrupt occurs, or when a previously running thread voluntarily relinquishes the processor, the process manager chooses another thread to run. It does so by first choosing the container, then requesting the container to choose the process. This two step process may be modified in situations involving gang or co-scheduling in shared memory multiprocessors, instead, one or more processes in the ready containers may be assigned to the processors. The reflective meta-level architecture of *μChoices* “opens up” the implementation and allows the dynamic customization of process scheduling at each of these two steps. This is accomplished by creating two parallel meta-hierarchies: the process manager meta-hierarchy and the process container meta-hierarchy.

5.1 Process Manager Meta-Architecture

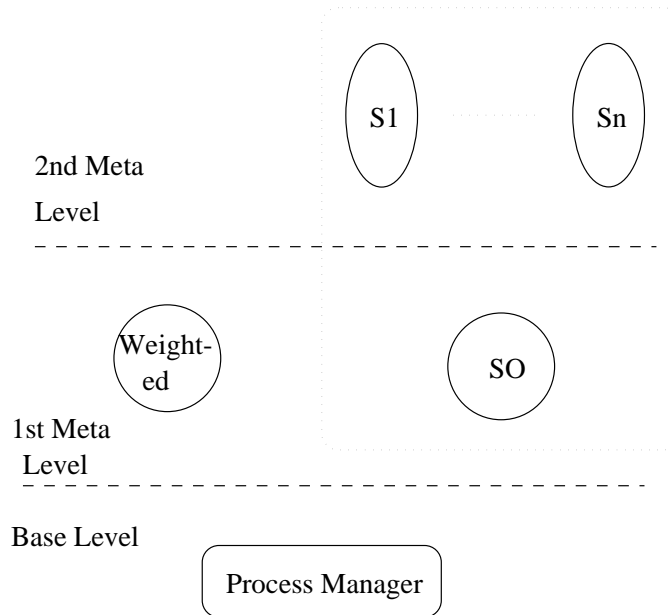


Figure 4: Meta-hierarchy of the process manager.

Figure 4 depicts the meta-hierarchy of the process manager. The semantic behavior of the process manager is defined by its set of process containers and its container selection algorithm. The meta-interface for the process manager is thus concerned with manipulating the set of container objects, and modifying the process manager’s selection algorithm. The meta-interface affords two ways of changing the status quo, as well as a method for querying the current status of the process manager.

- **SetContainer**

Containers are specified by container indices in the process manager. This method allows one to add, modify or delete a process container at a given container index. Two examples using this interface is given in figure 5. Process containers are associated with a time slice (units are in clock ticks) and non-negative *weight*. The default *Weighted* selection scheme always selects the first non-empty container with the highest weight.

- **GetContainer**

```

SetContainer ‘‘Operation=add; Type=Fifo; TimeSlice=50000; Weight=3;’’
SetContainer ‘‘Operation=delete; QueueIndex=2’’

```

Figure 5: Example `SetContainer` meta-operations.

This method accesses container attributes at specified container indices. Selection algorithms use this method to inspect containers and select containers based on their changing run-time attributes (eg. a selection algorithm which always chooses the largest container must determine the number of *Process* instances in each container).

- **SetSelectionPolicy**

This method allows the selection of container selection policy. The default meta-object defines a *Weighted* selection policy. A `SchedulingScriptingPolicy` selection policy allows custom selection through an embeddable script. Scripts are manipulated through a unified second level meta-interface.

5.2 Process Container Meta-Architecture

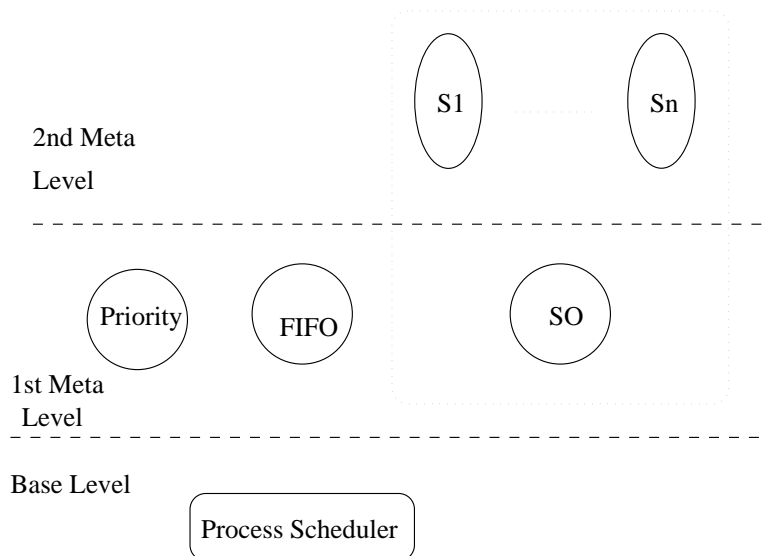


Figure 6: Meta-hierarchy of the Process Container.

Figure 6 depicts the process container meta-hierarchy. Process containers are repositories of processes. Containers act like schedulers: they contain a list of processes. We open up the simple list organization of a process container. Processes are always added to the rear of the list. Thus a scheduling policy only needs to find the proper process to *remove*. The first meta-level interface in this hierarchy gives methods for controlling the removal policy through the `SetSchedulingPolicy` method.

The *ProcessContainer* class exports an interface which allows the manipulation of the list by the scheduling policy meta-object. The interface allow clients to “walk” down the list of processes. Those methods include:

- **Initialize**

Perform initialization before the list is accessed.

- **GetCurrentProcessPriority**
Return the priority of current process.
- **PointerMoveTo**
Move the pointer in the process ready queue to given location in the list.
- **RemoveThisProcess**
Remove the process at currently pointed position.
- **LastProcess**
Return the pointer which points to the last process of this process ready queue.
- **NullPointer**
Return true if current pointer points to the null location.

5.3 Fault Detection and Handling

A faulty script can cause the process management system, and thus the entire operating system, to malfunction. In order to prevent this occurrence, we limit the way a script can affect the kernel environment.

First, as the process manager is driven by time slice interrupts, scripts cannot change the interrupt state of the kernel. Second, interpretive environments provide restricted interfaces to executing scripts. The environments for process manager and process container scripts offer different access points.

Process manager scripts may only inspect the contents and attributes of the manager's containers. If a container is not selected by the script, the process manager is not able to schedule a process. However, since the script cannot affect interrupts, the process manager is invoked at the next time slice interrupt. If the manager discovers that the script did not complete, it considers the script faulty and reverts to the basic *Weighted* selection policy. The scheme is thus *self healing* with respect to faulty selection scripts.

Process container scripts, on the other hand, may only access their own containers, through functions, such as **PointerMoveTo**, that were described in the previous section. A faulty removal script can cause, at worst, the corruption of its own container. Scripts that fail in their execution and do not return a process for scheduling are detected at the next time slice interrupt by the process manager. The container is discarded and all processes in the container are aborted.

5.4 Implementation Details

The current prototype implementation uses a Tcl interpreter. A separate interpreter is created for each scripting object. Each interpreter has a μ *Choices* system process executing the script in a read-eval-print loop.

Invoking a script thus involves first embedding the script into the interpreter through the script meta-interface. In order to execute the script, a context switch is performed to the interpreter's locus of execution. This context switch is lightweight, since no change of virtual memory domains occur in the kernel. The interpreter then runs the script, and returns the result through a context switch back to the primary invoking client.

	Default Policy	New Policy
Context Switching Time (μs)	1822	5946

Table 1: The context switching time of the default versus a scripted selection policy. The test was done on a Sun Sparcstation 10 with *VirtualChoices*.

We measured the time required for executing a context switch in the process management system with a selection policy dictated by a script. The test was done on a Sun Sparcstation

10 with μ VirtualChoices[30], a version of μ Choices that runs as a user process on top of UNIX. The results (table 1) indicate that the context switch time for the scripted policy is three times that of the default policy, mainly because of the extra two context switches into and out of the Tcl interpreter.

The overhead incurred by a Tcl implementation is high due to the cost involved in *invoking* the interpreter from within the kernel environment. The Java programming environment, on the other hand, allows users to pass *execution contexts* into the interpreter when it is invoked from the outside. No context switch is necessary as the execution context carries the information necessary to start script interpretation. Our preliminary results indicate that an untuned Java implementation required 8 milliseconds for a C++ to Java call. We have managed to reduce the overhead required for calling into Java to 2 microseconds by prefetching class name bindings.

6 Conclusion

Long-lived operating systems must accommodate flexible functionality and tune their performance to application behavior. The systems require structured dynamic customization facilities in order to remain manageable and efficient. We developed a meta-level architecture for dynamic customization based on scripting meta-objects. The semantics of scripting meta-objects depend on the scripts they execute, thus scripts are the meta-objects for the scripting meta-object itself. The base/meta separation of scripts from scripting objects ensures that a clear meta-interface exists for running, adding, updating and deleting scripts from scripting objects.

Our experiment with Tcl showed that it is possible to build a reflective framework for process management based on the scripting meta-level architecture. We decomposed the process management subsystem into two parallel meta-hierarchies, both of which are scripted.

Performance in a scripting framework requires a fast scripting language. Tcl does not meet this requirement. The structure of the current Tcl interpretive environment requires a context switch to an interpreter thread as well as a context switch back out. Previous work on scripting in operating system kernels have involved the use of “little languages” that keep interpretive overhead low. In designing the Jetstream LAN, Edwards et al.[6] amortize system call overhead over several operations by *batching* operations into a script. The design of the *Choices* memory object cache interface was driven by the intent to use a small language that could describe actions in a distributed shared virtual memory coherency protocol[25]. The reflective architecture in this paper generalizes these approaches through the use of scripting objects that interpret scripts in a flexible, general purpose language. Our preliminary experiences with the Java programming environment have been encouraging. The use of user-supplied execution contexts make context switching to the interpreter unnecessary. We have found that prefetching class and method information blocks in Java reduces the external invocation overhead even further. We expect that “just-in-time” compilation from portable byte-code to native machine code will give scripts the efficiency necessary for fine grained, time critical system tasks.

Scripts have several advantages over compiled object code. Scripts do not require trusted compilers or the sharing and exchange of cryptographic keys. An object-oriented scripting language allows the dynamic extension and incremental specialization of extension code itself. Scripts bring other flavors of flexibility to operating systems. We are implementing *active capabilities*, where a capability is composed, in part, of an encrypted script. The script can encode non-conventional access rights which are determined by executing the script.

The process of customization can be made more effective if the system under consideration can be monitored. In other work, we built an open model for visualizing the behavior of object-oriented operating systems[27]. Such a system may be used to control interactively and reconfigure the behavior of a running operating system. We are designing a dynamic visualization of a process subsystem. Such a visual presentation can help “see into” the system and assist the process of understanding and reconfiguration.

References

- [1] T. Anderson. The case for application-specific operating systems. In *Third Workshop on Workstation Operating Systems*, April 1992.
- [2] T. Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler Activations: Effective Kernel Support for the User Level Management of Parallelism. *ACM Transactions on Computing Systems*, 10(2):53–79, February 1992.
- [3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, bers C. Cha, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating System Principles*, December 1995.
- [4] Roy H. Campbell and See-Mong Tan. μ Choices: An Object-Oriented Multimedia Operating System. In *Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995. IEEE Computer Society.
- [5] David Cheriton. The V Distributed System. *Communications of the ACM*, pages 314–334, 1988.
- [6] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. *SIGCOMM '94*, August 1994.
- [7] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.
- [8] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Comm. of the ACM*, 34(9):28, September 1991.
- [9] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Sun Microsystems Computer Company, 2550 Garcia Avenue, Mountain View, California 94043. <http://www.sun.com>, May 1995.
- [10] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. Thirteenth ACM Symp. on Operating System Principles*, page 68, Pacific Grove, CA, October 1991. Published as Proc. Thirteenth ACM Symp. on Operating System Principles, volume 25, number 5.
- [11] N. Islam. *Customized Message Passing and Scheduling for Parallel and Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.
- [12] N. Islam and R. H. Campbell. Techniques for Global Optimization of Message Passing Communication on Unreliable Networks. In *International Conference on Parallel and Distributed Systems*, September 1995.
- [13] Nayeem Islam and Roy Campbell. “Uniform Co-Scheduling using object-oriented design techniques”. In *International Conference on Decentralized and Distributed Systems*, September 1993.
- [14] G. Kiczales. Foil For The Workshop On Open Implementation. In *Workshop for Open Implementation '94*, Salishan Lodge, Gleneden Beach, Oregon, October 1994.
- [15] Swee-Boon Lim. *Adaptive Caching in a Distributed File System*. PhD thesis, University of Illinois at Urbana-Champaign, November 1995.
- [16] Peter W. Madany, Roy H. Campbell, and Panos Kougiouris. Experiences Building an Object-Oriented System in C++. In Jean Bezivin and Bertrand Meyer, editor, *Technology of Object-Oriented Languages and Systems Conference*, pages 35–49. Prentice Hall, Paris, France, March 1991.
- [17] Michael J. McLennan. [incr tcl] - Object-Oriented Programming in TCL. In *Tcl 93 Workshop*, 1993.

- [18] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Usenix Mach Workshop*, pages 17–29, October 1990.
- [19] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.
- [20] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. 3rd Int'l. Conf. on Distr. Computing Sys.*, page 22, October 1982.
- [21] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, December 1995.
- [22] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):12–32, 1988.
- [23] Richard Rashid. Threads of a New System. *UNIX Review*, 1986.
- [24] S. J. Leffler and M. K. McKusick and M. J. Karels and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1988.
- [25] Aamod Sane and Roy H. Campbell. Interfaces as Little Languages. Internal Memo, University of Illinois at Urbana-Champaign, Department of Computer Science, December 1992.
- [26] M. Sefika. A Dynamically Reconfigurable Application Programming Interface for an Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, Dec 1993.
- [27] M. Sefika and R. H. Campbell. An Open Visual Model for Object-Oriented Operating Systems. In *Fourth International Workshop on Object-Oriented Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society Press.
- [28] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *USENIX Technical Conference*, San Diego, California, January 1996.
- [29] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.
- [30] S. M. Tan, D . K. Raila, W. S. Liao, and R. H. Campbell. Virtual Hardware for Operating System Development. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, September 1995. <http://choices.cs.uiuc.edu/srg/stan/vchoices.ps>.
- [31] See-Mong Tan, David Raila, and Roy H. Campbell. An Object-Oriented *Nano-Kernel* for Operating System Hardware Support. In *Fourth International Workshop on Object-Oriented Operating Systems*, Lund, Sweden, August 1995. IEEE Computer Society.
- [32] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the 1992 International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 1992.
- [33] M. W. Young. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Technical Report CMU-CS-89-202, Carnegie Mellon University, November 1989.