# Specialization Tools and Techniques for Systematic Optimization of System Software

DYLAN McNAMEE, JONATHAN WALPOLE, CALTON PU, CRISPIN COWAN, CHARLES KRASIC, ASHVIN GOEL, and PERRY WAGLE
Oregon Graduate Institute of Science & Technology
and
CHARLES CONSEL, GILLES MULLER, and RENAULD MARLET
University of Rennes/IRISA

Specialization has been recognized as a powerful technique for optimizing operating systems. However, specialization has not been broadly applied beyond the research community because current techniques, based on manual specialization, are time-consuming and error-prone. The goal of the work described in this paper is to help operating system tuners perform specialization more easily. We have built a specialization toolkit that assists the major tasks of specializing operating systems. We demonstrate sthe effectiveness of the toolkit by applying it to three diverse operating system components. We show that using tools to assist specialization enables significant performance optimizations without error-prone manual modifications. Our experience with the toolkit suggests new ways of designing systems that combine high performance and clean structure.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Performance

Additional Key Words and Phrases: Operating system specialization, optimization, software architecture

## 1. INTRODUCTION

A key dilemma for operating systems designers is to reconcile the apparently conflicting requirements of correct operation across all applications and high performance for individual applications. The conventional approach to address this dilemma is to write code that is general-purpose, but

Authors' addresses: D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, and P. Wagle, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 200000 NW Walker Road, Beaverton, OR 97006; C. Consel, G. Muller, and R. Marlet, University of Rennes/IRISA.

optimized for a few anticipated common cases. However, the result is an implementation with performance characteristics that are fixed throughout the lifetime of the operating system. Problems arise when common cases vary from installation to installation and grow worse when they vary dynamically.

Much of the operating systems research in the past decade has investigated alternative approaches to addressing these diverse requirements. A promising approach is to incorporate customizability into system structure [Bershad et al. 1995; Engler et al. 1995; Massalin and Pu 1989; Rashid et al. 1989; Small and Seltzer 1994]. A customizable operating system can be tuned for the currently observed common conditions. In most implementations of this approach, the ability to be customized is designed into the operating system, but the actual customized code is written by experts and manually injected into the system. We call such techniques explicit customization. In addition to enabling optimizations for specific common cases, explicit customization can be used to extend the functionality provided by the system. The drawbacks of *explicit customization* are (1) that significant burden is placed on system tuners, and (2) optimization opportunities are reduced because customizable systems explicitly limit access to global system state. Some optimizations require access across module boundaries, which is often prohibited to provide safety. Furthermore, the advantages of the approach are not provided to legacy applications or to applications that are unwilling or unable to take on the responsibility for tuning the operating system to their needs.

An alternate approach, *inferred customization*, is based on automatically deriving optizations instead of writing them by hand. We advocate an approach to inferred customization based on *specialization*. Specialization improves the performance of generic operating system code by creating optimized code for common cases. Specialization in this context consists of *restricting*[1] code for improved performance, rather than extending it, as is often the case with explicit customization.

Because specialized components are restrictions derived from the original system, the following benefits are possible: the burden on the system tuner is reduced, and specialized components can take advantage of any state in the system. Furthermore, the benefits of a specialized system are transparently provided to legacy as well as new applications. However, our experiences with applying specialization manually [Pu et al. 1995] have raised a number of drawbacks which have limited its adoption as a system-building technique. These drawbacks include that performing specialization correctly involves complex analysis of the system; hence, generating specialized code can be tedious and error-prone, and can result in systems which are more complex and harder to debug and maintain than the original. This paper introduces a toolkit we have developed that addresses these drawbacks by reducing the amount of manual work required to specialize operating systems.

[1]Using the term specialization to mean restriction has the opposite meaning when it is applied to class inheritance in object-oriented programming.

This paper is organized as follows. Section 2 describes the fundamentals of specialization: specialization predicates, partial evaluation, and guards. Section 3 describes the specialization toolkit. Section 4 presents our experiences with using the toolkit to specialize three areas of system code: signal delivery in Linux, the Berkeley packet filter interpreter, and Sun RPC. We describe the process of specializing each system component, and discuss the associated performance improvements. Section 5 discusses the strengths and weaknesses of the current toolkit based on our experiences. Section 6 presents work related to the various components of the specialization toolkit. Finally, Section 7 describes ongoing work on the specialization toolkit and summarizes our experiences.

## 2. AN OVERVIEW OF SPECIALIZATION

Specializing a system starts with a set of *specialization predicates*, which are states of the system that are known in advance. For example, the fact that two different variables, $x$ and $y$, have the same value could be a specialization predicate. *Partial evaluation* takes a generic source program $P_{gen}$, plus a set of specialization predicates that apply to it. Based on the specialization predicates the partial evaluator divides $P_{gen}$, into *static* and *dynamic* parts. The static part is evaluated at specialization-time, and the results are *lifted* to the output program. The remaining dynamic parts are combined with the lifted static output to form the specialized program, $P_{spec}$, in a procedure called *residualization* [Consel and Danvy 1993; Consel and Noël 1996; Sestoft and Zamulin 1988; Berners-Lee et al. 1996, no. 474]. The output of partial evaluation is an optimized system that relies only on dynamic inputs. We have identified three kinds of specialization, based on when specialization predicates become available.

—***Static specialization*:** When specialization predicates are known at compile-time, partial evaluation can be applied before the system begins execution. Section 4.1 describes our experience with applying static specialization to Sun RPC. The benefit of static specialization is that the costs of specialization are not incurred at run-time. The drawback is that it cannot take advantage of specialization predicates whose values are not established at compile-time.

—***Dynamic specialization*:** Deferring specialization until run-time allows the use of specialization predicates whose values are not established until some point during system execution, but which once established hold for the remainder of the execution. Section 4.2 describes our experience with applying both static and dynamic specialization to the BSD packet filter interpreter [McCanne and Jacobson 1993].

—***Optimistic specialization*:** Optimistic specialization extends the previous techniques to optimize the system for specialization predicates that only hold for bounded time intervals.[2] This kind of specialized code

[2]We have called these predicates *quasi-invariants* in previous papers.

optimistically assumes that the specialization predicates hold, and thus the rest of the system must ensure they do. Section 4.3 describes our experience with applying optimistic specialization to signal delivery in Linux. Optimistic specialization can take advantage of either statically or dynamically specialized code.

The main design issues related to specialization are correctness and performance. Specialization is a correctness-preserving transformation only if the specialization predicates hold while executing the specialized code that relies on them. For static specialization, specialization predicates cannot be violated once established. Dynamic and optimistic specialization predicates may not hold until some point during execution, so the specialized code needs to be enabled only when the specialization predicates are known to hold. Furthermore, optimistic specialization predicates may be violated as the system executes. Any time a specialization predicate is violated, executing the specialized code would result in erroneous behavior. Therefore any system event that can modify a specialization predicate term should disable the specialized code that relies on it. We use *guards* to enable and disable specialized code when specialization predicate terms are modified.

The time required to generate and insert the specialized code counts against the performance benefits of executing it. Thus, specialization is a net performance benefit only if the performance improvements of executing the specialized code outweigh its costs. This may require multiple executions of the same specialized code. For these reasons, specialization is most often applied to code segments that are executed a large number of times for the same specialization predicate (e.g., reads of a large file or long network stream).

The performance trade-offs of optimistic specialization are slightly more complicated. Efficient optimistic specialization is a matter of *moving* interpretation to the right place, since it requires additional checks to ensure that specialized code can be executed only when its specialization predicates hold. Thus the cost of executing those checks also counts against the benefits of specialization. Minimizing this additional cost often means moving the checks away from the location of the specialized code to the locations where its specialization predicates are modified. The reasoning behind this approach is that effective specialization usually requires the specialized code to be frequently executed, and hence its location is a poor choice for guard placement.

## 3. A TOOLKIT FOR SPECIALIZING OPERATING SYSTEMS

We have developed a toolkit and methodology for specializing operating systems. A system tuner takes the following steps to specialize a system using our toolkit:

(1) ***Identify specialization predicates:*** There are three substeps to identifying specialization predicates in an operating system. The first is to

use the kernel developer's knowledge of the system's structure and environment to postulate that some predicate in the system is useful for specialization. The second is to locate code that can be optimized when the postulated specialization predicate holds. The third is to estimate the net performance improvement of taking advantage of the specialization predicate by comparing the specialization overheads to its benefits.

(2) ***Generate specialized code:*** Given a set of specialization predicates, specialized code can be generated for the system components that reference them. While this has been done by hand for small, isolated routines, it can be automated and scaled to larger system components using partial evaluation. In some cases, we have found it necessary to slightly restructure the code in order to enable partial evaluation.

(3) ***Check when specialization predicates hold:*** Dynamic and optimistic specialization are based on specialization predicates that do not necessarily hold when the system starts executing. Therefore, these systems need to detect when the specialization predicates are established and when they are violated.

The specialization predicates used by static and dynamic specialization are never violated once established. However, optimistically specialized code depends on specialization predicates not changing, and will be incorrect if any of them do change. To preserve correctness, the system tuner must locate all the places in the system that can cause specialization predicates to change, and *guard* them with code that will respecialize the affected components to reflect the new state of the specialization predicates. A guarded write to a specialization predicate term first triggers respecialization if a specialization predicate will be modified, then performs the modification.

(4) ***Replace specialized code:*** When specialized code is enabled (e.g., when a dynamic specialization predicate is established), or must be disabled (e.g., when an optimistic specialization predicate is changed), the system must replace one version of the code with another. We call this operation *replugging*. Since many operating systems are concurrent programs, the current version of the code may be *in use* when replugging occurs. Therefore, some form of synchronization between invoking specialized code and the replugger is required. Ideally, the overhead of this synchronization should be incurred during replugging and not while invoking the specialized code, since the former executes less often than the latter. Therefore an asymmetric synchronization mechanism for replugging is appropriate.

The specialization toolkit helps system tuners with steps two, three, and four. For step one, we have found no tool that can substitute for the kernel developer's intuition for locating feasible specialization predicates. However, system-profiling tools [Anderson and Berc 1997; McVoy and Staelin 1996; Silicon Graphics 1999; Tamches and Miller 1999; Yaghmour 1999]

could be used to estimate, via dynamic execution counts, the benefits of repeated execution of the specialized code versus the overheads of performing the specialization. Our toolkit does not currently contain such a profiler; hence, they are not discussed further in this paper. For step two, the Tempo partial evaluator is our tool for generating specialized code. The TypeGuard and MemGuard tools are used in step three to help system tuners locate code which may modify specialization predicates and enable or disable specialized code appropriately. Finally, the Replugger is our tool for safely replacing specialized code in the face of concurrent execution in step four. The rest of this section describes these tools in detail.

### 3.1 Tempo: Specialized Code Generator

Tempo is a partial evaluator for C programs [Consel and Danvy 1993; Jones et al. 1993]. The main challenge for partial evaluation is to separate, given a set of specialization predicates, the static parts of the program from the dynamic parts. The analysis phase that performs this separation is called binding-time analysis.

A significant challenge for binding-time analysis is to deal with C language features such as pointers, structures, and functions with side-effects. Pointers and aliases are problematic because they make it difficult to prove that the dynamic parts of the program do not affect the static parts. Structures and arrays are problematic because their values do not have a textual representation, and therefore the partial evaluator has no way of lifting them into the residualized program even if they are static. Finally, functions that modify global (e.g., heap-allocated) state have side-effects on any other functions that access that state. The presence of such side-effecting functions can cause binding-time analysis to declare all functions that access the global state to be dynamic. Each of these limitations reduces the accuracy of binding-time analysis, leaving more of the program as dynamic, and not specializable. More significantly, since dynamism propagates transitively, conservative analyzers applied to such code often declare entire programs to be dynamic, even in the presence of specialization predicates.

Unfortunately, operating systems code makes heavy use of pointers, arrays, structures, and side-effecting functions. Tempo was designed to cope with the aspects of C usage that are common in operating systems. Another challenge is that the long-lived execution of operating systems means that predicates that may be useful for specialization for some periods of time may be dynamic in others. Conventional approaches to binding-time analysis do not capture, or ignore, this situation. Our use of Tempo, particularly in the context of optimistic specialization, was specifically designed to address the long-lived nature of operating system code.

Tempo's binding-time analysis is more accurately able to determine which parts of a program are static than previous binding-time analyses. This feature is important because it dramatically increases the potential for specializing the system. Tempo has features that address the problems

of conventional binding-time analyses when applied to operating system code written in C [Muller et al. 2000]:

—***Use sensitivity:*** enables an accurate treatment of nonliftable values, i.e., values which do not have textual representation such as pointers, structures, and arrays [Hornof et al. 1997]. Tempo's approach to treating nonliftable values computes a distinct binding time for each variable's use, depending on its context.

—***Flow sensitivity:*** enables a single variable to be static following a static definition and dynamic otherwise. Tempo associates a unique binding time each assignment of a variable and dependent reads of that variable [Hornof and Noyé 1997]. In contrast, flow-insensitive binding-time analyses conservatively consider variables that are assigned to both static and dynamic values to be dynamic.

—***Context sensitivity:*** Interprocedural binding-time analysis is crucial for operating systems code. More specialization opportunities can be exploited when the analysis computes a description for each calling context (e.g., different binding-times associated to the same argument). A conservative binding-time analysis considers a given function argument to be dynamic if there exists a call to this function where this argument is dynamic. In contrast, Tempo's context-sensitive analysis assigns a specific binding-time description for each binding-time call context [Hornof and Noyé 1997].

—***Return sensitivity:*** Many functions in operating systems have dynamic side-effects (e.g., to modify global state). Such side-effecting functions cannot be statically evaluated. However, in some cases the return value of a side-effecting function can be determined to be static (e.g., a successful return status). Previous binding-time analyses make such a function dynamic. Tempo's return-sensitive binding-time analysis enables the return value of a function to be static even when its body needs to be residualized. Return sensitivity enables more code in the calling function to be determined static, thus enabling more specialization to occur.

Tempo can perform both compile-time and run-time specialization. To optimize the performance of run-time specialization, Tempo generates, at compile-time, a dedicated run-time specializer and object-code templates with holes for the values of the static computations. At run-time, the specializer performs the static computations, selects the templates representing the dynamic computations, and fills in their holes with the static values.

### 3.2 Enabling and Disabling Specialized Code

By definition, specialized code can only be correct when its specialization predicates hold. Thus the system tuner needs to ensure that specialized code is kept consistent with the state of specialization predicates. For static

specialization, this step is trivial because the specialization predicates are invariant. For dynamic and optimistic specialization, however, the system tuner needs to ensure that specialized code is not enabled before the specialization predicates are established. Further, with optimistic specialization, the specialization predicate can be violated after being established, so the specialized code must be disabled (or the system respecialized) when a specialization predicate term is modified.

Dynamic and optimistic specialization predicates are established by *binding phases* in systems. Binding phases can be explicit or implicit. Examples of explicit bindings are a client establishing an RPC connection to a server and a process opening a file for reading. Examples of implicit bindings are inferring a relationship between two processes based on observing repeated signals sent between them and inferring sequential file access by observing repeated read calls without intervening seeks. System tuners need to locate the code that creates and destroys bindings related to specialization predicates. The observation that enables automatic location of binding and unbinding events is that both kinds of events modify specialization predicate terms.

We have designed two tools to help system tuners locate the code that establishes or destroys specialization predicates. The first tool, TypeGuard, operates on the program source code and uses type information to locate sites that should be guarded. The second, MemGuard, uses memory protection mechanisms to identify modifications to specialization predicate terms. The two tools are discussed in detail in the following subsections.

3.2.1 *TypeGuard*. There are two obvious methods of ensuring that a specialization predicate holds. One is to test it every time it is used (read). The other is to test it every time it is modified (written). For specialization to be efficient, the specialization predicate terms must be used more frequently than they are modified. Based on this observation, our approach to solving the guarding problem is to place guards at the site of modifications to specialization predicate terms.

Accurately locating modifications to specialization predicate terms is nontrivial. Consider a simple example in which a specialization predicate term is a global variable. A naive solution might be to simply search through the source code for occurrences of the variable's name using a tool such as `grep`. There are two problems with this approach, however. The first is that it may report too many sites to guard because different variables with the same name may be locally defined within functions. The second is that it may not report all of the sites that need guarding, due to aliases. For example, the specialization predicate term may be passed by reference to a function that modifies it via a different variable name.

To further complicate matters, many useful operating system specialization predicate terms are not simple scalar global variables, but are fields of dynamically allocated structures. This characteristic not only highlights the problem of dealing with aliases, but also introduces the need to distinguish among instances of the same structure type. To illustrate these

issues, consider the following example from the Linux kernel's signal delivery code which checks whether the signalling process is owned by the same user as the process to be signaled. The specialization predicate

```
current->uid == p->uid
```

refers to the `uid` field of two specific instances of type `task_struct`. These two pointers could be aliases for the same structure, or two different structures. Furthermore, elsewhere in the code, different aliases could be used to refer to these instances. The only textual representation common among aliases to these instances is the name of the type they point to, `task_struct`.

To address these challenges, we use a two-phase approach for detecting modifications to specialization predicate terms. The first phase performs a static analysis to identify the structure types whose fields are specialization predicate terms. These types are then extended to include an additional specialization predicate ID (`SPID`) field. The first phase also identifies all statements that update a guarded field and inserts the guarding code that performs the second phase. The second phase involves dynamically setting the `SPID` field when specialized code is enabled, clearing it when specialized code is disabled, and checking it when a specialization predicate term is modified.

This type-based approach detects modifications made to structure fields that are specialization predicate terms as long as the modification is made via the containing structure. The C language allows the creation of aliases that point directly to guarded fields, which creates capabilities for specialization predicates to be modified without going through their containing structures. To prevent these capabilities from allowing unguarded writes to specialization predicate terms, we extend the first phase to flag the operations that create them, which include

—type-casted assignment from or to the guarded type,

—attempting to guard a field that is part of a union, and

—taking the address of a field that is guarded.

When a statement causes a warning, the system tuner must either remove the offending statement by restructuring the code, or examine subsequent uses of the flagged value to manually assure that either the capability does not modify the specialization predicate, or that those modifications are guarded.

As an example of guarding, the assignment `current->uid = bar` would be written as

```
if (current.SPID!= NULL)
   current.SPID->update_uid(bar);
else
   current->uid = bar;
```

For specialization predicate terms, the `update_uid` function invokes the replugging procedure and writes the `current->uid field`, which we describe in Section 3.3.

This guarding code identifies structure instances that are *not* specialization predicates with one additional memory reference and comparison against `NULL`. If the structure does contain specialization predicate terms, it invokes the code necessary to evaluate the continued validity of the specialization predicate. The guard code is not currently inserted automatically, but it is sufficiently simple that it can be packaged inside a simple macro that can be inserted by hand. However, it could easily be automated. Our implementation of TypeGuard is based on the SUIF compiler toolkit [Amarasinghe et al. 1995].

3.2.2 *MemGuard: Testing Guard Coverage*. In the absence of a type-safe language, any type-based guarding tool cannot *guarantee* complete coverage. Our approach to this problem in TypeGuard is to issue warnings about alias-producing operations. These must currently be validated by hand. It would be useful to automate the verification required when TypeGuard issues warnings. In addition, there are opportunities for modifying specialization predicate terms that occur in operating systems independent of the language's type safety, such as passing pointers to device drivers or assembly routines. We have built a tool that can guarantee complete guard coverage. This tool, MemGuard, uses memory protection hardware to write-protect pages that contain specialization predicate terms. The write-fault handler checks if the address being written is a specialization predicate term, and if so, performs a guarded write which triggers replugging when needed. By using hardware memory protection, MemGuard is guaranteed to capture *all* writes to specialization predicate terms.

The main drawbacks of this approach are that memory protection hardware has coarse-granularity and high overheads. The granularity of protection is virtual memory pages, so modifications to any data that share a page with a specialization predicate term will result in false hits. The cost of false hits is high because the performance of a memory write to any location on a guarded page is reduced by a factor of about 1,000 [Cowan et al. 1997].

Even though these high overheads make it inappropriate for production use, MemGuard can be used as an effective tool for debugging software guard placement. Performance of executing guarded writes is greatly improved by extending the software guard code to disable the hardware memory protection as it modifies the specialization predicate term. In this case, the only memory protection traps caught by MemGuard would be caused by either false hits or code which should have a software guard. False hits could be eliminated, at the cost of memory consumption, by laying out guarded structures on two adjacent pages, with the guarded fields on one page, and the unguarded fields on another. Running a system with MemGuard through a set of kernel test suites would be an effective

way of automatically ensuring software guards are placed everywhere a specialization predicate can be modified.

### 3.3 Replugger: Dynamic Function Replacement

When a dynamic specialization predicate is established or an optimistic specialization predicate is violated, the system must replace the current code with code that is consistent with the new state of the specialization predicate. This replacement operation is called *replugging*. In our approach, replugging is performed at the granularity of C functions, and refunctions are invoked via indirect function pointers.

Maintaining correctness during replugging is nontrivial because the current version of the function may be in-use when the replugging operation occurs. An obvious way to preserve correctness in the face of concurrent replugging is to use locks to synchronize function invocation and replugging. However, this approach may add unacceptable performance overheads to the invocation of specialized functions. For this reason, we desire an asymmetric synchronization mechanism in which the overhead of invocation is as low as possible, at the potential expense of additional overhead for replugging, since specialized code is usually invoked more often than it is replaced.

Two factors affect the design of a correct replugging mechanism:

(1) Whether concurrent invocation of the same repluggable function is possible.

(2) Whether there can be concurrency between replugging and invocation.

The first factor, concurrent invocation, is affected by the scope of repluggable functions. A designer can avoid concurrent invocation by associating repluggable functions with threads. Doing this makes replugging simpler because there can be only one invoking thread at a time, and enables specialization predicates associated with thread state. Sharing repluggable functions among multiple threads may save code space, but it also makes replugging more complex. One reason for this additional complexity is that concurrent invoking threads should not execute different versions of the same function.

The second factor, concurrency between replugging and invocation, can occur on either uniprocessors or multiprocessors. On a uniprocessor, it can happen if an invoking thread can block inside a repluggable function, thus allowing a replugger to execute. On a multiprocessor, replugging threads can run concurrently with invoking threads. Another way that replugging and invocation can be concurrent is for an interrupt-level event to invalidate a specialization predicate. Since the replugging operation must wait for any pending invocation to complete, handling interrupt-level repluggers correctly is complex, and we have avoided this possibility by not using specialization predicates that are modified at interrupt-level.

In the case without concurrency among either invocation or replugging, no special mechanism is required—it is correct for the replugger to simply

Table I. Overhead for Synchronizing Invocation and Replugging (cycles)

| | Reader-Writer Spinlocks | Counting Replugger | Boolean Replugger |
|---|---|---|---|
| Invocation overhead | 48 | 60 | 4 |
| Replugging overhead | 50 | 340 | 340 |

update the function pointer. For the cases with concurrency, the replugger must not install a new function until all threads executing in the previous one have exited. In order to detect whether threads are executing a repluggable function, a counter is incremented on invocation and decremented on return. Once replugging has begun, new threads must not be able to invoke any version of the function being replugged. To achieve this goal, the replugger replaces the previous version of the function by a stub function, called `holding_tank`. The replugging thread blocks if any threads are actively invoking the previous version of the function. When a thread enters `holding_tank` it blocks on a condition variable until replugging has completed. As the last thread exits the previous version of the function, it decrements the count to zero and signals the replugging thread. Replugging completes when the replugging thread wakes up, replaces the `holding_tank` function pointer with the new function, and signals the threads in the `holding_tank` to start executing the replugged function.

This version of the replugger, which accommodates both concurrent replugging and invocation, is called the counting replugger. In the case without concurrent invocation (e.g., specialized code is bound to threads), we have simplified the counting replugger's counter to a boolean flag. Using a flag significantly reduces the overhead of invocation by replacing bus-locked arithmetic on the counter with atomic memory reads and writes.

Table I compares the performance of our counting and boolean repluggers to a standard reader-writer spinlock on a two-processor 450MHz Pentium II running Linux 2.2.14. The measured times, in cycles, demonstrate the desired asymmetric performance characteristics, particularly of the boolean version of the replugger, in which only four cycles are added to the invocation path. The four cycles consist of setting the flag, clearing the flag, and a branch not taken to wake up a waiting replugger.

Further details about the operation and implementation of an HP/UX version of the replugger are described in Cowan et al. [1996], and the code for the Linux version of the replugger is available at `www.cse.ogi.edu/sysl/projects/synthetix`.

## 4. EXPERIMENTS

In order to evaluate the effectiveness of our specialization toolkit, we applied it to a wide range of operating system components. This section describes our experiences with using the tools to specialize three disparate system components: marshaling in Sun RPC [Sun Microsystems 1988b], interpreting Berkeley Packet Filter (BPF) programs [McCanne and Jacobson 1993], and the Linux signal delivery mechanism.

### 4.1 Specializing Remote Procedure Calls

Remote procedure call (RPC) is the basis for NFS [Sun Microsystems 1988a], NIS [Ramsey 1994; Sun Microsystems 1999], and other Internet services. At the heart of Sun RPC is the eXternal Data Representation (XDR) standard which is a machine-independent format for passing RPC parameters. The process of translating into and out of XDR is called *marshaling*. Marshaling is a key source of overhead in RPC. Marshaling is performed by stubs, which are generated automatically from an Interface Definition Language (IDL) specification. The `rpcgen` program takes as input the IDL specification, and produces stubs and header files which are compiled and linked into programs that use RPC. When a client makes an RPC call, the stub for that call marshals the data into a message buffer and sends the message to the server. The server-side stub demarshals the data and invokes the server routine called by the client. The results of invoking the routine are marshaled and sent back to the client. The client stub completes the process by demarshaling the results and returning from the remote call.

The RPC stubs are composed of a set of microlayers, each devoted to a small task. For example, there are layers to read and write data during marshaling and to manage the exchange of XDR-encoded messages through the network. This section reports our experience applying Tempo to the marshaling stubs, the output of which was compiled and linked into the RPC client and server [Muller et al. 1997; 1998]. The specialization predicates we used in this experiment are available when the stubs are generated, and are never violated. Thus, this is an example of static specialization.

4.1.1 *Specialization Opportunities in Sun RPC.* Sun RPC's marshaling code uses data structures to hold state associated with the binding between an RPC client and server. Some fields of those structures have values that are known at stub generation time, and the computations that depend only on these values can be performed statically. The resulting specialized marshaling code consists of only the computations that depend on the dynamic values. In contrast, the generic marshaling code repeatedly interprets and propagates the values of both static and dynamic fields through the layers.

The following sections illustrate some specific specializations in Sun RPC's marshaling stubs using code excerpts that are annotated to show the static and dynamic computations derived by Tempo's binding-time analysis. In the following figures, dynamic computations are printed in **bold**; static computations are printed in `roman`.

*Eliminating encoding/decoding dispatch.* Sun RPC's dispatch of encoding and decoding uses a form of interpretation that is amenable to specialization. The generic function `xdr_long` (see Figure 1) is capable of both encoding and decoding long integers. It selects the appropriate operation to perform based on the field `x_op` of its argument `xdrs`. For encoding, `x_op`

```
bool_t xdr_long(xdrs,lp)  // Encode or decode a long integer
   XDR *xdrs;             // XDR operation handle
   long *lp;              // pointer to data to be read or written
{
   if (xdrs->x_op == XDR_ENCODE) // If in encoding mode
    return XDR_PUTLONG(xdrs,lp); // Write a long int into buffer
   if (xdrs->x_op == XDR_DECODE) // If in decoding mode
    return XDR_GETLONG(xdrs,lp); // Read a long int from buffer
   if (xdrs->x_op == XDR_FREE) // If in "free memory" mode
    return TRUE;   // Nothing to be done for long int
   return FALSE;   // Return failure if nothing matched
}
```

Fig. 1. Reading or writing a long integer: `xdr_long()`.

== XDR_ENCODE. For decoding, x_op == XDR_DECODE. These are the specialization predicates.

Specialization reduces the function xdr_long to three different functions[3]—one per static value of x_op—each of which consists of a single return statement which is inlined, removing the function call altogether.

*Eliminating buffer overflow checking*. Another form of interpretation appears when buffers are checked for overflow. This situation applies to the function xdrmem_putlong, shown in Figure 2. As parameter marshaling proceeds, the remaining space in the buffer is maintained in the field x_handy. The marshaling code initializes x_handy to the initial buffer size, which is a constant determined by the stub generator. Each call to xdrmem_putlong decrements x_handy by sizeof(long) and tests it for negative value (corresponding to buffer overflow). Tempo is able to statically determine whether x_handy's value ever falls below zero because the initial value (BUFSIZE) and the decrement value (sizeof(long)) are both specialization predicates, and because it can statically count the total number of calls. Thus, the specialized function consists of either the buffer copy or a statically generated exception if an overflow is discovered at specialization time.

*Propagating exit status*. The third example extends the optimizations enabled by the specialization predicates used in the previous examples. The return value of the procedure xdr_pair, shown in Figure 3, depends on the return value of xdr_int, which in turn depends on the return value of xdr_putlong. After specialization, both xdr_int and xdr_putlong have static return values.[4] Thus the return value of xdr_pair is known at specialization-time. Tempo propagates this known return value to the caller of xdr_pair (i.e., clntudp_call, not shown here), so xdr_pair no longer needs to return a value, and its return type becomes void. The specialized function, with the specialized calls to xdr_int and xdr_putlong independently, is shown in Figure 4. Tempo has determined that the

[3]There is an additional value of x_op and associated function for "free memory" mode.
[4]Note, that despite its static return value, the function xdr_pair has side-effects, and thus is not static.

```
bool_t xdrmem_putlong(xdrs,lp)                  // Copy long int into output buffer
   XDR *xdrs;                                  // XDR operation handle
   long *lp;                                   // pointer to data to be written
{
   if((xdrs->x_handy -= sizeof(long)) < 0)     // Decrement space left in buffer
    return FALSE;                              // Return failure on overflow
   *(xdrs->x_private) = htonl(*lp);            // Copy to buffer
   xdrs->x_private += sizeof(long);            // Point to next copy location in buffer
   return TRUE;                                // Return success
}
```

Fig. 2. Writing a long integer: xdrmem_putlong().

```
bool_t xdr_pair(xdrs, objp) {          // Encode arguments of rmin
   if (!xdr_int(xdrs, &objp->int1))    // Encode first argument
    return (FALSE);                    // Possibly propagate failure
   if (!xdr_int(xdrs, &objp->int2))    // Encode second argument
    return (FALSE);                    // Possibly propagate failure
   return (TRUE);                      // Return success status
}
```

Fig. 3. Encoding function xdr_pair().

```
void xdr_pair(xdrs,objp) {             // Encode arguments of rmin
                                       // Overflow checking eliminated
   *(xdrs->x_private) = objp->int1;    // Inlined specialized call
   xdrs->x_private += 4u;              // for writing the first argument
   *(xdrs->x_private) = objp->int2;    // Inlined specialized call
   xdrs->x_private += 4u;              // for writing the second argument
                                       // Return code eliminated
}
```

Fig. 4. Specialized encoding function xdr_pair().

```
Xdr_vector(XDR *xdrs, char *basep, u_int nelem, u_int elemsize) {
   register u_int i;
   register char *elptr;

   elptr = basep;
   for (i = 0; i < nelem; i++) {
     if (! Xdr_int(xdrs, (int *) elptr)) {
       return(FALSE);
     }
     elptr += elemsize;
   }
   return(TRUE);
}
```

Fig. 5. Marshaling an array: xdr_pair().

return value is always TRUE independently of the dynamic objp argument. Propagating this return value to the body of the caller eliminated another comparison, not shown here.

*Marshaling loop unrolling*. When an RPC argument is an array, the marshaling code iterates over the array, marshaling each element in turn. Often, the length of a marshaled array is known in advance, and can be used as a specialization predicate term. Figure 5 shows the code for marshaling an array. In this code, the number of elements, nelems, and

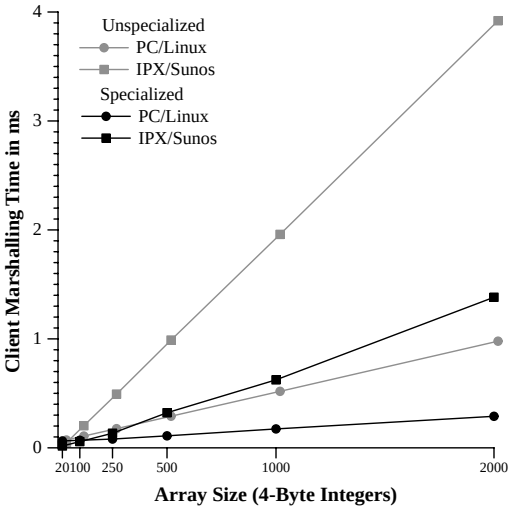


Fig. 6. Client marshaling overhead comparison. The gray lines are unspecialized client marshaling overheads; the dark lines are the corresponding specialized overheads.

the size of each element, `elemsize`, are both specialization predicate terms. This enables Tempo to automatically unroll the for loop in `Xdr_vector`, which eliminates `nelem` additions, comparisons, and branches.

4.1.2 *Performance Results*. We analyze the performance improvements of specialization by evaluating its impact on marshaling overhead alone, as well as the overall round-trip RPC times. Two experimental platforms were used to gather measurements. The first consisted of two Sun IPX 4/50 workstations with 64KB of cache running SunOS 4.1.4, with a 100Mbit/sec ATM network connection. The second consisted of two 166MHz Pentium PCs with 512KB of unified L2 cache running Linux, with a 100Mbit/sec ethernet connection.

*Marshaling overhead improvements*. Figure 6 illustrates the impact of specialization on marshaling latency. For most message sizes, specialization reduces latency by more than a factor of two. On the PC, this speedup increases linearly with the amount of data marshaled. This behavior is expected because the number of instructions eliminated by specialization is linear with the message size. However, we found that on the Sun IPX the speedup decreases as the amount of marshaled data increases. The reason for this unexpected behavior is that on this platform execution time is dominated by memory accesses, not instruction execution. As the data to be marshaled grow, a larger portion of the marshaling time is spent copying the data into the output buffer. While specialization decreases the number of instructions used to marshal data, the number of memory accesses remains constant. Therefore the savings due to specialization become less significant as the message size grows.

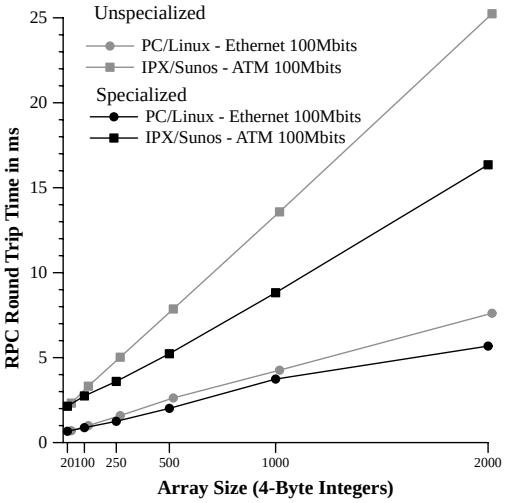


Fig. 7. RPC round-trip comparison. The gray lines are unspecialized RPC latencies; the dark lines are the corresponding specialized latencies.

Table II. Size of the SunOS Binaries (in bytes)

| | Message Size | | | | |
|---|---|---|---|---|---|
| Client Code | 20 | 100 | 500 | 1000 | 2000 |
| Generic | | | 20004 | | |
| Specialized | 24340 | 27540 | 33540 | 63540 | 111348 |

*Round-trip RPC improvements*. In order to evaluate the effect of improved marshaling latency on overall RPC performance, we measured average round-trip RPC latencies for arguments that consist of various sized arrays. Figure 7 shows a 55% improvement in round-trip time on the Sun/IPX platform and a 35% improvement on the PC/Linux platform. One reason these improvements are less than for marshaling alone is because of the cost for initializing message buffers on both client and server sides. This cost, combined with network access latency, reduces the net performance improvements. However, even with these mitigating factors, specialization has a significant positive impact on round-trip RPC performance.

4.1.3 *Code Size*. Table II shows the effects of specialization on code size. Because of loop unrolling, the size of the specialized RPC marshaling code grows with message size, and can greatly exceed that of the generic code. This increase in code size can affect cache performance, although in our experiments (shown in Figures 5 and 6), any degradation of cache performance was dominated by the improvements of specialization. We performed an additional experiment on the PC to evaluate the cache performance impacts of loop unrolling. Table III shows the performance of two versions of the specialized code. The first version fully unrolls loops; the second version limits unrolling to 250 iterations, which we found fits in the

Table III. Specialization with Loops of 250-Unrolled Integers (time in ms)

| Message Size | PC/Linux | | | | |
|---|---|---|---|---|---|
| | Original | Specialized | Speedup | 250-Unrolled | Speedup |
| 500 | 0.29 | 0.11 | **165%** | 0.108 | **170%** |
| 1000 | 0.51 | 0.17 | **200%** | 0.15 | **240%** |
| 2000 | 0.97 | 0.29 | **235%** | 0.25 | **290%** |

PC's level two cache. The times shown were derived by dividing elapsed time by the total number of round-trip iterations. The results show that limiting loop unrolling improves the performance of the specialized code by approximately 10%. Further investigation of the cache impact of specialization, while interesting, is beyond the scope of this paper.

### 4.2 Specializing Packet Filters

The BSD Packet Filter (BPF) [McCanne and Jacobson 1993] provides a programmable interface for selecting packets from a network interface. The interface allows user applications to download packet filter programs written in a bytecode into a kernel- or library-resident packet filter interpreter. The bytecode programs decide when a packet matches the user's criteria. Matching packets are forwarded to the application. The `tcpdump` application prints network packets that match a user-defined predicate. This predicate is provided on the command line, and `tcpdump` translates it into a BPF program. The TCP packets that match the predicate are returned to `tcpdump`, where they are printed. The Linux kernel used in this experiment implements packet filters in the `libpcap` library. Other kernels, such as NetBSD, implement packet filters within the kernel.

4.2.1 *Specialization Opportunities in Packet Filter Interpretation*. Since a single BPF program is likely to be executed many times to examine many thousands of packets, it is an ideal candidate for specialization. In this case, the code being specialized is the packet filter interpreter, and the specialization predicates are derived from a particular packet filter program. Specializing an interpreter with respect to a particular program effectively compiles that program [Jones et al. 1985]. This is a case of either static or dynamic specialization, since the specialization predicate is never modified, once established. We measured two cases. First, in the static specialization case, the packet filter program is available well in advance of its execution. An example of this case is the static packet filter program used by `rarpd`, which selects RARP packets from network streams. Second, we considered dynamic specialization, in which the packet filter program is presented immediately before execution, and thus the overheads of specialization are included in the overall run-time. The `tcpdump` program is an example of this case, since the packet filter programs are generated from command-line input.

```
while(true) {
  switch (pc->opcode) {
    case LD:
      // do load instruction
    case JGT:
      if (accumulator > index_register)
        pc = pc->target_true
      else
        pc = pc->target_false
      // etc...
    case RET:
      // return instruction
      result =...
      break;
  }
  pc++
}
```

Fig. 8. Basic loop for BPF interpreter.

A session begins when an application hands the packet filter bytecodes to `libpcap` by calling `pcap_setfilter`. The application initiates filtering by calling `pcap_loop`. In the filter-loop, a packet is read and filtered by calling the `bpf_filter` function:

```
u_int bpf_filter(struct bpf_insn *pc, u_char *c,
  u_int wirelen, u_int buflen);
```

The parameters are the packet filter program, a packet, the length of the original packet, and the amount of the packet's data present. Of these parameters, the packet filter program is always the same during a session, so we derive specialization predicates from it. The `buflen` argument could also be used as a specialization predicate term, but in our experiments it was not because we wanted to focus on the benefits of eliminating interpretation.

The basic structure of the BPF interpreter is shown in Figure 8. The interpreter consists of an infinite loop, each iteration of which fetches the instruction pointed to by `pc`, uses a case statement to decode and execute the instruction, and finally increments `pc`. In addition, the interpretation of some instructions, such as jumps, modifies `pc` within the loop. When interpreting conditional jump instructions, such as `JGT`, the value assigned to `pc` depends on dynamic interpreter state. This approach to structuring the interpreter, as a case statement within an infinite loop, is problematic because it propagates the dynamism of `pc` throughout, making the interpreter unspecializable.

An alternate approach to building an interpreter, which is amenable to specialization, is to use recursion. In this approach, the while loop is replaced by a tail-recursive function which gets called for each new value of `pc`, as shown in Figure 9. We manually restructured the BPF interpreter using recursion in order to perform the experiments described below.

4.2.2 *Performance Results*. To evaluate the impact of specialization on the performance of interpreting packet filter programs, we specialized the

```
case JGT:
  if (accumulator > index_register)
    return(bpf_filter(pc->target_true, c, wirelen, buflen))
  else
    return(bpf_filter(pc->target_false, c, wirelen, buflen))
```

Fig. 9. Using recursion to make pc static.

Table IV. Specialized BPF Performance (time in seconds)

| Program | Run Time | Run Time—Null Filter |
|---|---|---|
| Nill | 2.60 | NA |
| Original, unspecialized | 4.34 | 1.74 |
| Statically specialized | 2.84 | 0.24 |
| Dynamically specialized | 3.35 | 0.75 |

interpreter for a simple packet filter program which counts packets. We compared this specialized program to the unspecialized interpreter on the same packet filter program. We wanted to isolate the benefits of specialization from the unavoidable overheads of the packet filter mechanism. We did this by constructing a null packet filter, which incurs the unspecializable overheads of the packet filter mechanism, but without performing any packet filter interpretation.

Table IV presents the execution times to filter 10 megabytes of ethernet packets. We measured the null packet filter and three versions of the counting packet filter: the original unmodified version, a statically specialized version, and a dynamically specialized version. The dynamically specialized version includes the overhead of executing the run-time specializer to generate the specialized code. In addition, the statically specialized code is more efficient than the dynamically generated template-based specialized code. The right column isolates the packet filter interpretation cost by subtracting the execution time of the null filter. In both the static and dynamic cases, specialization yields significant performance improvements.

4.2.3 *Code Size*. Partial evaluation unrolls the fetch, decode, execute loop of the BPF interpreter, thus raising the possibility of impact on code size. Common packet filters are between five and 15 instructions. The unspecialized interpreter has 550 lines. The interpreter specialized for a six-instruction filter is 366 lines. For a 10-instruction filter, the specialized version is 576 lines. These results show the code size impact is small for common-size packet filter programs.

### 4.3 Specializing Signals

UNIX signals are a mechanism for communicating events between processes. The statement:

```
ret = kill (pid, n);
```

causes process *pid* to suspend its current activity and run a procedure designated as the handler for signal *n*. Figure 10 shows the structure of the `kill` system call in Linux. The function `sys_kill` is the kernel-side entry

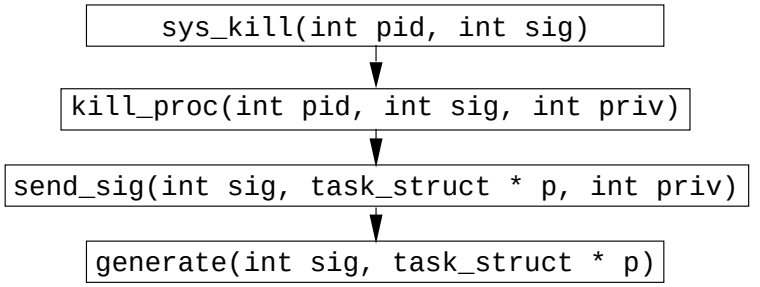


Fig. 10. Linux `kill` system call architecture.

point of the kill system call. The function `kill_proc` searches the process table for the `task_struct` of the process being signaled which is specified by `pid`. The function `send_sig` checks for errors and valid permissions. The function generate interprets the signal number, and delivers the signal to the destination process by setting state in `task_struct`, and calls `wake_up_process` if needed. The source code for these functions in the unspecialized Linux kernel (version 2.0.27) is shown in Figure 11.

When a process repeatedly sends the same signal to the same destination process, it is likely that the fields of both processes' `task_structs` are unchanged. This observation presents an opportunity for optimistic specialization: these fields can be guarded, and used as specialization predicate terms for the generation of specialized signal delivery code.

The difficulty in applying specialization in this way is that the relationship between the communicating processes is not represented explicitly in the code, and hence must be inferred through observations. For example, if a process sends the same signal to the same target twice, we might infer an ongoing communication relationship. The problem with applying this approach to the signal code in Linux is that no history is maintained between invocations. Therefore, to allow specialization we added a field, `last_sig_to`, to each process' `task_struct` to cache information about the last signal it sent.

The rewritten version of the `sys_kill` function compares the values of `sig` and `pid` to those in `last_sig_to` to verify that the signal is indeed a repeat. If it is, `sys_kill` invokes signal delivery code that is specialized to send this particular signal between this pair of processes. Our rewritten versions of `send_sig` and `generate` cache the signal number and the identities of the source and destination of the signal for subsequent executions to be able to detect repeated signals.

The specialized code for delivering the signal `SIGUSR1` in Figure 12 was generated by Tempo, based on the following specialization predicates (`current` is a pointer to the `task_struct` of the signal source, and `p` is a pointer to the `task_struct` of the signal destination):

```
current->last_sig_to == p
last_sig_to->uid == p->uid
last_sig_to->session == p->session
last_sig_to->euid == p->euid
```

```
static inline void generate(unsigned long sig, struct task_struct * p)
{
    unsigned long mask = 1 << (sig-1);
    struct sigaction * sa = sig + p->sig->action - 1;

    /*
     * Optimize away the signal, if it's a signal that can
     * be handled immediately (ie non-blocked and untraced)
     * and that is ignored (either explicitly or by default)
     */
    if (!(mask & p->blocked) && !(p->flags & PF_PTRACED)) {
       /* don't bother with ignored signals (but SIGCHLD is special) */
       if (sa->sa_handler == SIG_IGN && sig != SIGCHLD)
          return;
       /* some signals are ignored by default.. (but SIGCONT already did its deed) */
       if ((sa->sa_handler == SIG_DFL) &&
           (sig == SIGCONT || sig == SIGCHLD || sig == SIGWINCH || sig == SIGURG))
          return;
    }
    p->signal |= mask;
    if (p->state == TASK_INTERRUPTIBLE && (p->signal & ~p->blocked)) {
       wake_up_process(p);
    }
    if (!current) return;
    if (!current->pid) return;
    if (intr_count) return;

    switch (sig) {
    case SIGUSR1:
        sdl_replug_start(current->sp_kill_proc);     /*
        current->last_sig_to = p;                     * Guarded write to specialization
        current->last_sig = sig;                      * predicate terms
        p->last_sig_from = current;                   */
        sdl_replug_end(current->sp_kill_proc, kp_usr1);
        break;
    default:
        sdl_replug_start(current->sp_kill_proc);
        p->last_sig_from = current->last_sig_to = current->last_sig = NULL;
        sdl_replug_end(current->sp_kill_proc, yelp);
    }
}


int send_sig(unsigned long sig,struct task_struct * p,int priv)
{
    if (!p || sig > 32)
       return -EINVAL;
    if (!priv && ((sig != SIGCONT) || (current->session != p->session)) &&
        (current->euid ^ p->suid) && (current->euid ^ p->uid) &&
        (current->uid ^ p->suid) && (current->uid ^ p->uid) &&
        !suser())
       return -EPERM;
    if (!sig)
       return 0;
    /*
     * Forget it if the process is already zombie'd.
     */
    if (!p->sig) {
       sdl_replug_start(current->sp_kill_proc);       /*
       current->last_sig_to = p;                       * Guarded write to specialization
       current->last_sig = sig;                        * predicate terms
       p->last_sig_from = current;                     */
       sdl_replug_end(current->sp_kill_proc, kp_usr1);
       return 0;
    }
    if ((sig == SIGKILL) || (sig == SIGCONT)) {
       if (p->state == TASK_STOPPED)
          wake_up_process(p);
       p->exit_code = 0;
       p->signal &= ~( (1<<(SIGSTOP-1)) | (1<<(SIGTSTP-1)) |
            (1<<(SIGTTIN-1)) | (1<<(SIGTTOU-1)) );
    }
    if (sig == SIGSTOP || sig == SIGTSTP || sig == SIGTTIN || sig == SIGTTOU)
       p->signal &= ~(1<<(SIGCONT-1));
    /* Actually generate the signal */
    generate(sig,p);
    return 0;
}
```

Fig. 11. Unspecialized Linux `kill` system call source code.

These specialization predicates allow Tempo to eliminate most of the comparisons and conditionals in the signal delivery code, and directly do the work for delivering a signal in the body of `generate`.

```
int kill_proc(int pid, int sig, int priv)
{
    struct task_struct *p;

    if (sig<0 || sig>32)
       return -EINVAL;
    for_each_task(p) {
       if (p && p->pid == pid)
          return send_sig(sig,p,priv);
    }
    return(-ESRCH);
}

asmlinkage int sys_kill(int pid,int sig)
{
    int err, retval = 0, count = 0;

    if (!pid) return(kill_pg(current->pgrp,sig,0));

    if (current->last_sig_to && current->last_sig_to->pid == pid &&
          current->last_sig == sig && current->sp_kill_proc) {
       retval = (* sdl_executor(current->sp_kill_proc))();
       sdl_executor_end(current->sp_kill_proc); /* Invoke specialized kill_proc */
       return retval;
    }

    if (pid == -1) {
       struct task_struct * p;
       for_each_task(p) {
          if (p->pid > 1 && p != current) {
             ++count;
             if ((err = send_sig(sig,p,0)) != -EPERM)
                retval = err;
          }
       }
       return(count ? retval : -ESRCH);
    }
    if (pid < 0)
       return(kill_pg(-pid,sig,0));
    /* Normal kill */
    return(kill_proc(pid,sig,0));
}
```

Fig. 11. *Continued*.

```
int kp_usr1 ()
{
   struct task_struct *p;
   {
     if (((*(*current).last_sig_to).sig != (void *) 0) == 0) {
       send_sig_SSSDDDStr_sigaction_DDDSSSS_flat2 = 0; /* This is a Tempo-declared global integer */
       goto pprocfin0;
     }
     {
       struct sigaction *sa;
       unsigned int *suif_tmp2;

       sa = (struct sigaction *) ((char *)
          (*(*(*current).last_sig_to).sig).action + 160) - 1;
       suif_tmp2 = &(*(*current).last_sig_to).signal;
       *suif_tmp2 = *suif_tmp2 | 512;
       if ((*(*current).last_sig_to).state == 1 &&
            ((*(*current).last_sig_to).signal &
            ~(*(*current).last_sig_to).blocked) != 0u)
         wake_up_process ((*current).last_sig_to);

     }
     send_sig_SSSDDDStr_sigaction_DDDSSSS_flat2 = 0;
   pprocfin0: ;
   }
   return send_sig_SSSDDDStr_sigaction_DDDSSSS_flat2;
}
```

Fig. 12. Specialized `kill` system call source code: `kill_proc`, `send_sig`, and `generate` folded and specialized.

This specialization is optimistic because if any of the specialization predicate terms are modified between signals, the specialized code is invalid and must be replugged. For example, if the destination process exits

(thus invalidating the `last_sig_to` pointer), or if the `euid` or `uid` of the source or destination process is modified, the specialized version of the kill system call could either crash the machine by indirecting through an invalid `task_struct` pointer, or could produce incorrect results by sending a signal without permission

We used TypeGuard to identify locations that require guarding. Given the set of specialization predicate terms, TypeGuard produced a list of program statements that could modify those terms. TypeGuard includes in this list all of the program statements (such as typecasts) that could allow the specialization predicate term to be modified elsewhere. We manually inspected the locations identified by TypeGuard and when we determined they could modify specialization predicate terms, we placed a guard that replugged to the unspecialized function before performing the modification. Since Linux kernel threads are nonpreemptive, and these experiments were conducted on a uniprocessor, we used the boolean version of the replugger.

4.3.1 *Performance Results*. The latency of delivering a signal consists of a number of components. These components are: looking up the `task_struct` in the process table, updating the signaled process' state, saving the signaled process' context, entering and exiting the kernel, and the unpredictable scheduling delay incurred between making a process runnable and the time it starts executing the signal handler. Figure 13 shows these components for the specialized and unspecialized versions of signal delivery, but without the unpredictable scheduling delay. This scheduling delay was eliminated by measuring the latency of a process sending the signal `SIGUSR1` to itself, which causes the signal handler to be directly invoked upon return from the system call. For this experiment, one user was logged in, running an X11 server and three xterm programs, and a few other X11 applications, for a total of 62 processes, resulting in a table lookup time of 26.5 $\mu$-seconds. With a more intensive workload, the table lookup would take even more time. The specialized version of the system call avoids this lookup, thus eliminating its overhead. The work required to update the signaled process' state was reduced from 14.5 $\mu$-seconds to 12 $\mu$-seconds. Overall, the specialized system reduces the latency to send a signal by 65%. The size of the process table clearly has a major impact on the cost of the `kill` system call, but even in a situation without the table lookup overhead, specialization reduces execution time by 15%.

4.3.2 *Application-Level Impact*. The application-level impact of improving the performance of delivering signals depends on how applications use them. Signals are often used to signal exceptional information between processes, such as `SIGKILL` from a shell to halt the currently executing process. However, signals are also used to implement more general services. For example, Leroy's POSIX threads implementation for Linux [Leroy 1996] uses Linux's variable-weight processes with shared address spaces. This threads package uses signals to communicate between threads, e.g., to wake up blocked threads when a mutex is released.

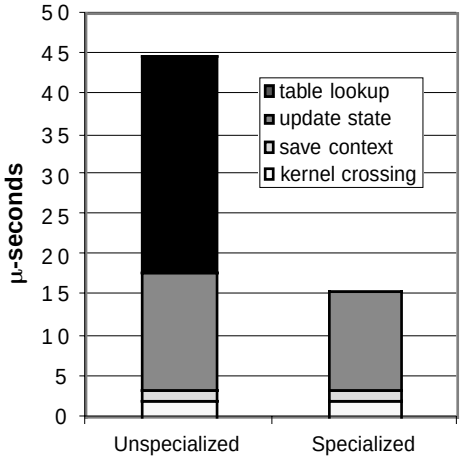


Fig. 13. Components of signal delivery latency: unspecialized vs. specialized.

Using this threads package, we wrote a test program that made extensive use of signals by synchronizing frequently. The test program is an implementation of the classic producer-consumer problem, using thread mutex's for synchronization. The test does 100,000 producer-consumer iterations with a buffer size of four items. Four executions of the test program on the unspecialized kernel had an average run-time of 11.9 seconds, with a standard deviation of 3.7 seconds. Four executions of the same program on the specialized kernel had an average run-time of 5.6 seconds, with a standard deviation of 0.7 seconds. The large variation in performance in the unspecialized code is caused by the random position of the target task in the process descriptor list. In the specialized code, the process table lookups are only performed the first time a signal is sent to a process; the subsequent lookups are specialized away. The remaining variability in the specialized code is due to nonscheduling behavior.

4.3.3 *Code Size*. The unspecialized signal delivery code, shown in Figure 11, has 59 lines of nonwhitespace code among four functions. Specializing this code eliminated the error checking on specialization predicate terms and folded the four functions into one. The resulting function, `kp_usr1`, shown in Figure 12, has 18 lines. The code required to guard the specialization predicate terms added a total of 60 lines among 11 functions in four files.

## 5. DISCUSSION AND EXPERIENCES

### 5.1 Results and Experiences with the Toolkit

The experiences reported in this paper represent examples of all three types of specialization: static, dynamic, and optimistic. We applied specialization to a range of system components, and reduced execution time by between 15% and 93%.

The experiments exercised our specialization toolkit across a wide range of system component types, each of which presented differing degrees of complexity. The remote procedure call experiment was the easiest to perform, since it was an instance of static specialization; hence, it required no dynamic enabling or disabling of specialized code. In addition the binding stages are clearly defined in the RPC protocol, thus making the specialization predicates easy to identify. This experiment demonstrated that Tempo can specialize complex machine-generated code, and achieve significant speedups.

The packet filter experiment was more difficult to perform, since the interpreter was written in a style that prevents specialization. Once the systematic modifications to the interpreter were made, this experiment showed that static specialization reduces interpretation time by a factor of seven. In addition, dynamic specialization, in which the overhead of generating the specialized code is counted against the benefits, reduces interpretation time by a factor of two. This experiment demonstrated Tempo's ability to specialize an entire domain-specific language interpreter, with a complex specialization predicate (an entire packet filter program).

The signal experiment was the most difficult of the three, since optimistic specialization includes guarding and replugging. In addition, the lack of explicit specialization predicates required that we modify the code to reify the state of the previous signal in order to detect repeated signals. This experiment exercised our guard placement tools as well as Tempo. We used TypeGuard to locate the kernel statements that potentially modify each of the specialization predicate terms. TypeGuard produced a large number of false positive reports,[5] mostly relating to allocating new structure instances, which do indeed modify specializapredicate terms, but do not violate specialization predicates, since newly allocated memory cannot contain specialization predicate terms. In contrast, the `free` operation applied to a specialization predicate term does need to be guarded, since after that operation the specialization predicate term no longer exists. We had to manually examine each of TypeGuard's reports to determine whether it required guarding. In addition to our guarding experiences, we found that optimistic specialization poses additional challenges to Tempo. The complication is that optimistic specialization predicate terms are actually modified by parts of the system. If those parts are given to Tempo, binding-time analysis will (accurately) determine the specialization predicates to be dynamic, and thus unspecializable. In order to account for the fact that the specialization predicates were guarded, we had to selectively omit parts of the code before presenting them to Tempo. Having done this, Tempo effectively specialized the signal delivery code, resulting in application-level performance improvements of a factor of two.

[5]For example, running TypeGuard on the Linux kernel looking for modifications to `task_struct_t.pid` resulted in 219 reports, yet only 44 of them were modifications to a specialization predicate term.

### 5.2 Lessons for System Tuners

As with any optimization, specialization is best applied to the common paths of an operating system. With these common paths identified, we found two kinds of system constructs that lend themselves to specialization.

The first construct to look for is session-oriented operations, such as file open/close, socket open/close, RPC binding, etc. In these situations, the binding stages are explicit, which eases the task of identifying specialization predicates. For example, the file open call establishes a binding between a file and a process. The specialization predicates resulting from this binding are related to the user's file permissions, the layout of the file on disk, whether the file is shared or not, etc. In addition, these explicit binding events directly trigger enabling and disabling code, which simplifies the task of placing guards.

The second important construct is domain-specific language interpreters or compilers used by a system. The execution behavior of language-based components is described by the domain-specific program. Examples of such constructs include using Java to extend web servers or clients [Gosling et al. 1996], active networks [Tennenhouse et al. 1997], and other mobile code systems [Adobe Systems 1984; White 1994; Woolridge and Jennings 1995]. When the same program is used repeatedly, it can be a useful specialization predicate. With the exception of self-modifying code, program-based specialization predicates are never modified, making them useful for static or dynamic specialization, and avoiding the overheads related to optimistic specialization.

### 5.3 Lessons for System Designers

The lessons for software architects designing a system from scratch are to employ the constructs that are amenable to specialization.

The first lesson is to make relationships between components explicit, rather than implicit, whenever possible. This encourages explicit sessions in place of implicit relationships. The evolution of the HTTP protocol [Berners-Lee et al. 1996; Fielding et al. 1999] exemplifies this trend: HTTP 1.0 created a short-lived TCP connection for each data request from a client to a server, while HTTP 1.1 utilizes persistent TCP connections between clients and servers. The latter approach is more likely to yield useful specialization predicates.

When explicit sessions are not appropriate, the next lesson is to be able to recognize imconnections from repeated patterns of actions. Recognizing patterns often requires additional state to be maintained across interactions. We found this to be useful in the signal example, where state was used to detect repeated signals between two processes, which allowed us to derive a specialization predicate.

Finally, using domain-specific languages, interpreters, and stub compilers is a powerful technique, not only because they raise the level of abstraction of system components, but also because they naturally give rise

to useful specialization predicates. We have begun investigating approaches to building software systems as layers of virtual machines and interpreters in this manner [Consel and Marlet 1998].

The main implication of our methodology is on system complexity and ease of maintenance. Making optimizations automatic and based on the source code allows the source code base to be left generic and easily understandable, modular, and maintainable.

## 6. RELATED WORK

### 6.1 Related Specialization Research

Our specialization-based approach to operating systems implementation is an instance of a programming methodology called *multistage programming* [Taha and Sheard 1997]. Multistage programming refers to programs that generate other programs, usually with the goal of improving performance. The program that generates or analyzes programs in a staged programming system is called a *metaprogram*, and the result is the *generated-program*. Staged programming techniques can be distinguished along a number of axes, including [Taha 1999]:

—***Automatic vs. manual annotation:*** Whether the static and dynamic portions of the input program are identified by an analysis (e.g., binding-time analysis) or via manual annotations (e.g., pragmas written by the system tuner).

—***Homogeneous vs. heterogeneous:*** Whether or not the generator (the metaprogram) is written in the same language as the generated program.

—***Static vs. run-time generation:*** Whether generated code is produced before run-time (static generation), or during run-time (run-time generation).

—***Two-stage vs. many-stage:*** If the output program is itself a metaprogram, the process can be applied recursively. Most such systems are also homogeneous, because it allows the transformations applied at each stage uniformly, which makes building them simpler, but excludes legacy code.

By this categorization, Tempo is an automatic tool, since its binding-time analysis automatically derives the dynamic and static labelings of program components. Tempo is a heterogeneous system, because its input is C-language programs, and its output is C and object code. Furthermore, its analysis core is written in ML. Tempo supports both static and run-time code generation. Tempo can be a two- or three-stage system. When performing static specialization, Tempo is two-stage. Tempo's dynamic specialization is three-stage, since it produces a generator that produces specialized code [Marlet et al. 1999].

C-Mix [Andersen 1993] is another partial evaluator for C programs that fits in the same staged programming categorization as Tempo. Like Tempo,

C-Mix can partially evaluate C programs, do interprocedural analysis, and deal with complex data structures and side-effects. However, it was not specifically designed to deal with systems code, and its analysis is not as precise as Tempo's. In particular, C-Mix is flow-insensitive, which means that a variable is considered dynamic as soon as it is dynamic in any part of the program, including exception handling. C-Mix also consumes more code space, because it eagerly replicates code to avoid problems in binding-time analysis. In contrast to Tempo and C-Mix, which operate on ANSI C programs, DyC [Grant et al. 1999] and 'C [Engler et al. 1986] are C-based languages that incorporate annotations to assist binding-time analysis to enable partial evaluation. DyC caches compiled code to improve the performance of dynamic specialization.

Other staged programming research projects use functional programming languages, such as ML. The MetaML language is a manual system that uses "staging annotations" instead of automatic analysis in order to make the resulting output more predictable [Sheard et al. 1999; Shields et al. 1998; Taha 1999; Taha and Sheard 1997]. Since MetaML input and output programs are in the same language as the staged programming system, MetaML is homogeneous, which enables it to support $n$-level staged programming. MetaML also supports both static and dynamic program generation.

In addition to the staged programming aspects of our work, there is work related to the other tools in the specialization toolkit. Lackwit [Callahan and Jackson 1997] is a program-understanding tool for C based on type inference. Unlike TypeGuard, which is based on C's types, Lackwit discards C's weak type system, and instead infers its own strong dynamic types for values based on the set of operations each value participates in, derived from a conservative data flow analysis of the program. Thus Lackwit can construct very specific types, e.g., the type of "pointers that are allocated and freed," as distinct from the type of "pointers that are allocated but *not* freed." This kind of analysis could be useful in placing guards for specialization predicates in system code, similar to TypeGuard. Lackwit performs more precise analysis than TypeGuard, but at the expense of using an algorithm that is exponentially complex in the worst case, which does not scale to the size of most systems code.

Tempo's binding-time analysis has similarities to the analyses used in program-slicing tools [Tip 1994]. Forward-slicing techniques propagate information from variable definitions to their uses, and have been used to define binding-time analyses for imperative programs [Das et al. 1995]. The analysis used by these tools is similar to the part of Tempo's binding-time analysis that propagates the state of variables from definitions to their uses. However, nonliftable values are not addressed by forward-slicing tools, and nor can they address values used in different contexts. Backward-slicing techniques propagate information from variable uses to their definitions. This is similar to the part of Tempo's binding-time analysis that computes the binding-time definitions of variables. The way Tempo computes the binding time of definitions is similar to backward

slicing which computes the commands that are needed in a slice. Unlike the two-point domain provided by slicing analysis (needed, not needed), Tempo's binding-time analysis is performed with a four-point domain (static, dynamic, static and dynamic, and {}), since some definitions may need to be both evaluated and residualized.

The Utah Flex project developed OMOS [Orr 1992], an object/metaobject server that supports the dynamic linking of executable modules. OMOS wraps dynamically instantiated execution modules in an object-oriented package, even if they were not written in an object-oriented language. OMOS provides considerably more functionality than our replugger, including the ability to specify which module should be loaded using certain code properties, such as whether it is in memory, or has been linked to sit at a particular address range. Thus OMOS encompasses some of the functionality of our specialization predicate guards, but does the checking only at load time.

### 6.2 Other Approaches to Operating System Customization

Aspect-oriented programming [Kiczales et al. 1997] provides a useful vocabulary for comparing approaches to system customization. An *aspect* of a system is a property which necessarily spans system components, and which is usefully considered independently. In this vocabulary, our methodology uses specialization to optimize the performance aspect of a system, and the guarding tools help ensure correctness when access to specialization predicate terms spans system components.

Aspect-oriented programming can be implemented using language tools built specifically to support it [Lopes and Kiczales 1998], or using a more general technique called *open implementations*, in which the implementation of a software module is tailorable by clients of that module [Kiczales 1996].

An open implementation of an operating system can be used to improve performance without altering functionality, or to implement additional functionality in an existing system. Building customizable operating systems using open implementations has been an active area of research in the last decade. Examples of such customizable operating systems include SPIN [Bershad et al. 1995], Exokernel [Engler et al. 1995], the Flux OSKit [Ford et al. 1997], Vino [Small and Seltzer 1994], SLIC [Ghormley et al. 1998], Choices [Campbell et al. 1992], and Apertos [Yokote et al. 1989].

In customizable operating systems correctness depends on extensions not being able to affect parts of the system beyond the extension's scope. SPIN provides such protection through the use of a type-safe programming language combined with a dispatcher which enforces constraints described by the service-writer [Pardyak and Bershad 1996]. For example, the dispatcher might enforce the constraint that a particular virtual memory extension can only handle faults for the process that installed it. SPIN also includes a hierarchical name-space that limits the scope of customized modules to only those tasks that specifically ask to use the customized

components. The responsibility of ensuring that customizations do not conflict with each other is left to extension-writers and the authors of built-in services.

Exokernel represents another approach to operating system customization. Exokernel pushes system services outside the kernel where they can be more easily and safely extended. As with SPIN, the responsibility of ensuring that customizations will not interfere with each other is left to the authors of the user-level system services and the developers of subsequent customizations.

The Utah Flux project has constructed a software architecture that supports replacement of operating system components, particularly *nesting* of operating system components [Ford et al. 1996; Ford and Susarla 1996] using the recursive virtual machine concept. Each virtual machine level can be customized for specific needs, and is protected from other virtual machines at the same level. The layers of indirection implicit in this structure come at some cost. However, specialization may be able to minimize these costs. The replaceable software components are large and complex, and the relationships between them will provide many specialization predicates because the components are not replaced frequently. We believe the modular structure of a system built with Flux could be particularly amenable to specialization, which would be one way to construct high-performance, highly structured systems.

Other researchers have used specialization in conjunction with specially designed system components to optimize a specific operating system service. For example, the Ensemble system uses specialization of network stacks written in ML in order to achieve high performance from modular components [Liu et al. 1999]. The Scout operating system achieves high performance by flattening network stacks automatically based on a new abstraction, paths, which are defined by programmers [Montz et al. 1995]. Scout uses domain-specific language compilers to produce optimized code from the path specifications.

## 7. CONCLUSIONS AND FUTURE WORK

This paper introduced concepts, tools, and a methodology for specializing operating systems code. We detailed the operation of tools for the specialization, guarding, and replugging phases of specialization. We evaluated the effectiveness of static, dynamic, and optimistic specialization by applying them in experiments that included more than one operating system (both Linux and Solaris), and a range of styles of code, ranging from regular system code (delivering signals), kernel-resident interpreters (interpreting packet filter programs), and a stub compiler (for RPC). These experiments demonstrated substantial performance improvements, comparable with those that are possible through hand-coded optimizations. Finally, we discussed the lessons we learned from these experiences and implications for future software engineering practices for system building.

The tools presented in this paper aid in the production of specialized code paths and in guarding them when they are used optimistically. Another important problem is how to identify good opportunities for specialization. In all our experiments to date, we have identified specialization opportunities by hand, using expert knowledge and heuristics to determine whether they would benefit from specialization. It would be useful to have tools to identify hot spots in operating systems, distill specialization predicates of such hot spots, and evaluate the feasibility of a given specialization strategy. There are many difficult specialization policy issues to solve such as whether a particular specialization is worthwhile given a particular guarding strategy and execution context, which specialized versions to generate ahead of time, which ones to cache, and what policies to use for managing such a cache.

One promising approach to addressing the complex trade-offs involved with the overheads and benefits of specialization is to dynamically monitor the run-time behavior of a system and analyze the net benefits of specializing individual system components given observed execution frequencies, and use feedback control to enable or disable specialization of each component in the system. We plan to use the microfeedback toolkit developed at OGI [Cen 1998; Goel et al. 1998] to develop controllers that achieve this kind of dynamic adaptivity of specialization policy in a way that is predictable, stable, responsive, and composable.

## ACKNOWLEDGMENTS

We would like to thank Julia Lawall for her insightful comments on multiple versions of this paper. We would also like to thank the TOCS reviewers for their careful critiques, which improved many aspects of our presentation. The authors were partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, and by grants from the Intel Research Council.

## REFERENCES

ADOBE SYSTEMS. 1984. *The PostScript Language Reference Manual*. 1st ed. Addison-Wesley, Reading, MA.

AMARASINGHE, S. P., ANDERSON, J. M., LAM, M. S., AND TSENG, C. W. 1995. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, Feb.). SIAM, Philadelphia, PA.

ANDERSEN, L. O. 1993. Binding-time analysis and the taming of C pointers. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-Based Program Manipulation* (PEPM '93, Copenhagen, Denmark, June 14–16), D. Schmidt, Chair. ACM Press, New York, NY, 47–58.

ANDERSON, J. M. AND BERC, L. M. 1997. Continous profiling: Where have all the cycles gone?. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP '97, Saint-Malo, France, Oct. 5–8), W. M. Waite, Ed. ACM Press, New York, NY.

BERNERS-LEE, T., FIELDING, R., AND FRYSTYK, H. 1996. Hypertext transfer protocol—HTTP/1.0. RFC 1945. ftp:// ftp.isi.edu/in-notes/rfc1945.txt.

BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E., FIUCZYNSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety, and performance in the SPIN operating

system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, Dec.). ACM Press, New York, NY, 267–284.

CAMPBELL, R. H., ISLAM, N., AND MADANY, P. 1992. Choices, frameworks and refinement. *Comput. Syst. 5*, 3, 217–257.

CEN, S. 1998. A software feedback toolkit and its application in adaptive multimedia systems. Ph.D. Dissertation. Oregon Graduate Institute of Science & Technology, Beaverton, OR.

CONSEL, C. AND DANVY, O. 1993. Tutorial notes on partial evaluation. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '93, Charleston, SC, Jan. 10–13), S. L. Graham, Ed. ACM Press, New York, NY, 493–501.

CONSEL, C. AND MARLET, R. 1998. Architecturing software using a methodology for language development. In *Proceedings of the International Symposium on Programming Language Implementation, Logics and Programs* (PLILP, Pisa, Italy). ACM, New York, NY.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '96, St. Petersburg Beach, FL, Jan. 21–24), H.-J. Boehm and G. Steel, Chairs. ACM Press, New York, NY, 145–156.

COWAN, C. AND MCNAMEE, D. 1987. A toolkit for specializing production operating system code. CSE-97-004. Oregon Graduate Institute of Science & Technology, Beaverton, OR.

COWAN, C., AUTREY, T., PU, C., AND WALPOLE, J. 1996. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems* (ICCDS '96, Annapolis, MD).

DAS, M., REPS, T., AND VAN HENTENRYCK, P. 1995. Semantic foundations of binding-time analysis for imperative programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (PEPM '95, La Jolla, CA, June 21–23), N. Jones, Chair. ACM Press, New York, NY, 100–110.

ENGLER, D. R., HSIEH, W. C., AND KAASHOEK, M. F. 1996. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '96, St. Petersburg Beach, FL, Jan. 21–24), H.-J. Boehm and G. Steel, Chairs. ACM Press, New York, NY, 131–144.

ENGLER, D., KAASHOEK, M., AND O'TOOLE, J. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (SIGOPS '95, Copper Mountain Resort, CO, Dec. 3–6), M. B. Jones, Ed. ACM Press, New York, NY, 251–266.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext Transfer Protocol—HTTP/1.1. RFC 2616. ftp://ftp.isi.edu/in-notes/rfc2616.txt.

FORD, B. AND SUSARLA, S. 1996. CPU inheritance scheduling. *ACM SIGOPS Oper. Syst. Rev. 30*, Winter, 91–105.

FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., AND SHIVERS, O. 1997. The Flux OSKit: A substrate for kernel and language research. *ACM SIGOPS Oper. Syst. Rev. 31*, 5, 38–51.

FORD, B., HIBLER, M., LEPREAU, J., TULLMANN, P., BACK, G., AND CLAWSON, S. 1996. Microkernels meet recursive virtual machines. *ACM SIGOPS Oper. Syst. Rev. 30*, Winter, 137–151.

GHORMLEY, D. P., PETROU, D., ANDERSON, T. E., AND RODRIGUES, S. H. 1998. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the 1998 USENIX Annual Technical Conference* (New Orleans, LA, June). USENIX Assoc., Berkeley, CA, 39–52.

GOEL, A., STEERE, D., PU, C., AND WALPOLE, J. 1998. SWiFT: A feedback control and dynamic reconfiguration toolkit. Oregon Graduate Institute of Science & Technology, Beaverton, OR.

GRANT, B., PHILIPOSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 1999. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design and Implementation* (PLDI '99, Atlanta, GA). ACM, New York, NY.

HORNOF, L. AND NOYÉ, J. 1997. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *SIGPLAN Not. 32*, 12, 63–73.

HORNOF, L., CONSEL, C., AND NOYÉ, J. 1997. Effective specialization of realistic programs via use sensitivity. In *Proceedings of the 4th International Symposium on Static Analysis* (SAS '97, Paris, France, Sept.). Springer-Verlag, Vienna, Austria.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ.

JONES, N. D., SESTOFT, P., AND SONDERGAARD, H. 1985. An experiment in partial evaluation: The generation of a compiler generator. In *Proc. of the first international conference on Rewriting techniques and applications* (Dijon, France, May 20-22), J.-P. Jouannaud, Ed. Springer Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 124–140.

KICZALES, G. 1996. Beyond the black box: Open implementation. *IEEE Software 13*, 1 (Jan.), 6–11.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP '97). Springer-Verlag, New York, NY. Lecture Notes in Computer Science, vol. 1241.

LEROY, X. 1996. The LinuxThreads library. http://pauillac.inria.fr/xleroy/linuxthreads/.

LIU, X., KREITZ, C., VAN RENESSE, R., HICKEY, J., HAYDEN, M., BIRMAN, K., AND CONSTABLE, R. 1997. Building reliable, high-performance communication systems from components. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP '97, Saint-Malo, France, Oct. 5–8), W. M. Waite, Ed. ACM Press, New York, NY.

LOPES, C. V. AND KICZALES, G. 1998. Recent developments inm AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP'98). Springer-Verlag, New York, NY. Lecture Notes in Computer Science, vol. 1445.

MARLET, R., CONSEL, C., AND BOINOT, P. 1999. Efficient incremental run-time specialization for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language and Design and Implementation* (PLDI '99, Atlanta, GA). ACM, New York, NY.

MASSALIN, H. AND PU, C. 1989. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, AZ, Dec. 3–6), G. Andrews, Chair. ACM Press, New York, NY.

MCCANNE, S. AND JACOBSON, V. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter Usenix Conference* (Jan.). USENIX Assoc., Berkeley, CA, 259–269.

MCVOY, L. AND STAELIN, C. 1996. lmbench: Portable tools fo performance analysis. In *Proceedings of the USENIX Technical Conference*. USENIX Assoc., Berkeley, CA.

MONTZ, A., MOSBERGER, D., O'MALLEY, S., PETERSON, L., PROEBSTING, T., AND HARTMAN, J. 1994. Scout: A communications-oriented operating system. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation* (OSDI '94, Monterey, CA, Nov.). USENIX Assoc., Berkeley, CA.

MULLER, G., MARLET, R., AND VOLANSCHI, E. N. 2000. Accurate program analyses for successful specialization of legacy system software. *Theor. Comput. Sci. 248*, 1-2.

MULLER, G., MARLET, R., VOLANSCHI, E. N., CONSEL, C., PU, C., AND GOEL, A. 1998. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems* (ICDCS '98, Amsterdam, The Netherlands, May). IEEE Computer Society Press, Los Alamitos, CA.

MULLER, G., VOLANSCHI, E.-N., AND MARLET, R. 1997. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. *SIGPLAN Not. 32*, 12, 116–126.

O'CALLAHAN, R. AND JACKSON, D. 1997. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering* (ICSE '97, Boston, MA, May 17–23), W. R. Adrion, Chair. ACM Press, New York, NY, 338–348.

ORR, D. 1992. OMOS—An object server for program execution. In *Proceedings of the International Workshop on Object-Oriented Operating Systems*.

PARDYAK, P. AND BERSHAD, B. N. 1996. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (OSDI '96, Seattle, WA, Oct. 28–31), K. Petersen and W. Zwaenepoel, Chairs. ACM Press, New York, NY, 201–212.

PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. 1995. Optimistic incremental specialization. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (SIGOPS '95, Copper Mountain Resort, CO, Dec. 3–6), M. B. Jones, Ed. ACM Press, New York, NY, 314–324.

RAMSEY, R. 1994. *All About Administering NIS+*. 2nd ed. Prentice-Hall, New York, NY.

RASHID, R. AND BARON, R. 1989. Mach: A foundation for open systems. In *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*. IEEE Press, Piscataway, NJ.

SESTOFT, P. AND ZAMULIN, A. V. 1988. Annotated bibliography on partial evaluation and mixed computation. In *Partial Evaluation and Mixed Computation*. North-Holland Publishing Co., Amsterdam, The Netherlands.

SHEARD, T., BENAISSA, Z., AND PASALIC, E. 1999. DSL implementation using staging and monads. In *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages* (DSL '99, Austin, TX). USENIX Assoc., Berkeley, CA.

SHIELDS, M., SHEARD, T., AND PEYTON JONES, S. 1998. Dynamic typing as staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '98, San Diego, CA, Jan. 19–21), D. B. MacQueen and L. Cardelli, Chairs. ACM Press, New York, NY, 289–302.

SILICON GRAPHICS. 1999. Lockmeter: Kernel spinlock metering for Linux IA32. Silicon Graphics, Inc., Mountain View, CA.

SMALL, C. AND SELTZER, M. 1994. VINO: An integrated platform for operating system and database research. Tech. Rep. TR-30-94. Department of Electrical Engineering and Computer Science, Harvard Univ., Cambridge, MA.

SUN MICROSYSTEMS. 1998a. NFS: Network File System protocol specification. Sun Microsystems, Inc., Mountain View, CA.

SUN MICROSYSTEMS. 1998b. RPC: Remote Procedure Call protocol specification, version 2. Sun Microsystems, Inc., Mountain View, CA.

SUN MICROSYSTEMS. 1999. *Solaris Naming Administration Guide*. Sun Microsystems, Inc., Mountain View, CA.

TAHA, W. 1999. Multistage programming: Its theory and applications. Oregon Graduate Institute of Science & Technology, Beaverton, OR.

TAHA, W. AND SHEARD, T. 1997. Multi-stage programming with explicit annotations. *SIGPLAN Not. 32*, 12, 203–217.

TAMCHES, A. AND MILLER, B. P. 1999. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (OSDI '99, New Orleans, LA., Feb.). USENIX Assoc., Berkeley, CA, 117–130.

TENNENHOUSE, D., SMITH, J., SINCOSKIE, D., WETHERALL, D., AND MINDEN, G. 1997. A survey of active network research. *IEEE Commun. Mag. 35*, 1, 80–86.

TIP, F. 1994. A survey of program slicing techniques. Tech. Rep. CS-R9438. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

WHITE, J. E. 1994. Telescript technology: The foundation for the electronic marketplace. General Magic, Inc., Mountain View, CA.

WOOLRIDGE, M. AND JENNINGS, N. 1995. Intelligent agents: Theory and practice. *Knowl. Eng. Rev. 10*, 2.

YAGHMOUR, K. 1999. Linux trace toolkit. http://www.info.polymtl.ca/home/karym/www/trace/.

YOKOTE, Y., TERAOKA, F., AND TOKORO, M. 1989. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of the 1989 European Conference on Object-Oriented Programming* (ECOOP '89, Nottingham, UK).

Received: March 2000; revised: January 2001; accepted: January 2001