January 1997

A toolkit for specializing production operating system code

Crispin Cowan

Dylan McNamee

Andrew Black

Calton Pu

Jonathan Walpole

See next page for additional authors

Author
Crispin Cowan, Dylan McNamee, Andrew Black, Calton Pu, Jonathan Walpole, Charles Krasic, Perry Wagle, Qian Zhang, and Renauld Marlet

A Toolkit for Specializing Production Operating System Code[*]

Crispin Cowan, Dylan McNamee, Andrew Black, Calton Pu, Jonathan Walpole
Charles Krasic, Perry Wagle, and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology [10pt]

Renauld Marlet
University of Rennes / IRISA

(synthetix-request@cse.ogi.edu)
http://www.cse.ogi.edu/DISC/projects/synthetix/

Abstract

Specialization has been recognized as a powerful technique for optimizing operating systems. However, specialization has not been broadly applied beyond the research community because the current techniques, based on manual specialization, are time-consuming and error-prone. This paper describes a specialization toolkit that should help broaden the applicability of specializing operating systems by assisting in the automatic generation of specialized code, and guarding *the specialized code to ensure the specialized system continues to be correct. We demonstrate the effectiveness of the toolkit by describing experiences we have had applying it in real, production environments. We report on our experiences with applying the tools to three disparate portions of operating systems: signal delivery, memory allocation and RPC. We describe how we used the toolkit to specialize these components, and present the resulting performance improvements. We conclude that a toolkit-based approach to specialization can work, and is an effective operating system optimization technique.*

1 Introduction

Specialization has been demonstrated to improve the performance of "generic" operating system code by dynamically creating optimized code for common cases that are discovered at run time [20, 26]. However promising, specialization has yet to make a significant impact outside the research community. This paper introduces a toolkit we are constructing that eases the task of specializing production operating system code. The toolkit enables the application of specialization techniques by a broader range of research and commercial operating system developers.

Our specialization toolkit addresses three difficulties with existing specialization techniques. First, manual specialization requires the hand coding of each special case. Our toolkit eases the task of building specialized systems by automatically generating specialized code and "guarding" the specialized code to ensure it is only executed when appropriate. Second, manual specialization often introduces global interdependencies when taking infrequently relevant code outside of the critical path. Our guarding tools make *composing* specialized modules feasible by isolating these global interdependencies. Third, manual specialization creates as many copies of the code as the number of special cases, making software maintenance expensive and error prone. Our tools ease the task of maintaining optimized operating system code by *preserving* the original source and managing the special cases for the programmer.

We have applied the specialization toolkit to a broad range of production systems software, including Linux signal delivery, the Vmalloc memory allocator and Sun's remote procedure call (RPC), yielding performance gains from 10% up to 1200%. These experiences and the resulting performance improvements demonstrate that tool-assisted specialization is an approach that improves operating system performance and at the same time, preserves system code maintainability and safety.

The remainder of this paper is organized as follows. Section 2 summarizes the major approaches to specialization and adaptive operating systems. Section 3 describes a specialization toolkit we have been developing at [institutions omitted for blind reviewing]. Section 4 presents our experiences with using the toolkit to specialize three areas of system code: signal delivery, memory allocation, and RPC. We describe the process of specializing each sys-

[*] This research is partially supported by DARPA grant N00014-94-1-0845 and DARPA contract F19628-95-C-0193, NSF grant CCR-9224375, and grants from the Hewlett-Packard Company and Tektronix.

tem component, then the measurements of performance improvements due to specialization. Section 6 discusses the strengths and weaknesses of the current toolkit based on our experiences. Finally, Section 7 describes ongoing work on the specialization toolkit and summarizes our results.

2 Customizing Operating Systems

Operating system specialization is a promising approach to improving application performance by *adapting* operating system behavior to individual application needs. This section is structured as follows. Section 2.1 describes the specialization-based approach to achieving operating system adaptivity. Section 2.2 describes the motivation for the tools, and describes the specialization subtask that each tool addresses. Finally, Section 2.3 describes how our tool-based specialization approach relates to other adaptive systems research.

2.1 Quasi-invariants and Specialization

Invariants are the building blocks for constructing specialized systems. We distinguish between two types of system invariants. A *true invariant*, like a classical invariant, is a state property of the system that is guaranteed to be true at all times. A *quasi-invariant* is a state property that is *likely* to remain true, but may become false at some future time. Either kind of invariant is stated as an expression using system variables that must evaluate to "true" to facilitate automatic exploitation of the invariant via specialization.

Given that some set of invariants and quasi-invariants are true, a specialized component can be created that improves functionality or performance over the generic component that it replaces. Optimizations can be done using formal, mechanical methods such as *partial evaluation* with respect to the invariants [8], or they can be at a higher level of changing the component's behavior while preserving the functional interface, such as changing the page replacement algorithm to adapt to application needs.

Previous specialization research has extensively explored filesystem operations, such as `read` [20, 26]. In these projects, various quasi-invariants related to kernel "open file" objects (file descriptors) were exploited as specialization opportunities. For example, when an application repeatedly performs small sequential reads, the file descriptor's current physical block number is a quasi-invariant. This quasi-invariant was used to generate a specialized version of `read` that performed better than the unspecialized `read` by more than a factor of 3 [26].

2.2 Motivation for a Specialization Toolkit

Even though specialization can be a powerful optimization technique, it has not been broadly applied in commercial operating systems. We believe this is because correct and effective specialization is hard to do. We have identified a number of causes for this:

- *Deciding* what *to specialize.* It is non-trivial to find opportunities for specialization. A variable may be modified in only one kernel procedure (thus suggesting candidacy for quasi-invariant status), but this procedure may be executed very frequently. Conversely, a variable may be modified by *many* procedures, but if those procedures live in "back roads" of the operating system, the variable could still be a good candidate quasi-invariant.
- *Generating specialized code.* The approach that has been used to specialize operating systems to-date is to manually generate specialized versions of procedures and write code that dynamically dispatches execution to these when deemed appropriate. This approach is problematic for three reasons. First, it is difficult to manually write code that fully exploits the invariance of a set of quasi-invariants. Logically, this task should be done by a compiler. Second, manually identifying the conditions for dispatching to a specialized routine can be tedious and error-prone. Third, identifying all the associated guards with respect to a quasi-invariant is difficult and error-prone, and missing any one of them would introduce a bug into the system.
- *Maintaining specialized code.* Current specialized systems are harder to maintain than their non-specialized counterparts. This is because parallel versions of code must be maintained: the generic version, and each of its specializations. Each modification to the generic version must be manually verified for its impact (or lack of impact) on each of the specialized versions. Furthermore, the system is complicated by the addition of code that dispatches between the generic and specialized versions of each specialized kernel procedure. Finally, changing the quasi-invariants also modifies the guards, which may be spread all over the system.

In order to build a toolkit that addresses these problems, we have decomposed the task of specializing an operating system into three components. (1) Discovering opportunities for specialization, (2) generating specialized code, and (3) ensuring correctness in the presence of specialization. We elaborate on these components in turn.

First, one of the most challenging aspects of specializing operating systems is discovering quasi-invariants that provide opportunities for effective specialization. One indication of the difficulty of this task is that all of the previous work in operating system specialization has concentrated on only one subsystem (the filesystem), and only a limited number of quasi-invariants within that subsystem (e.g., shared status, sequentiality of reads, and existence of holes in file layout.) [26]. The experiences reported in Section 4 indicate that tools can not replace human intuition and experience, but that tools *can* be used to assist kernel developers to evaluate quasi-invariant candidates, and aid the verification phase.

Second, after a set of quasi-invariants have been identified, the next task is to *partially evaluate* the system, by effectively recompiling the appropriate routines with the new assumption that what was previously assumed to be variable is now quasi-invariant. Previous specialization experiments involved manual partial evaluation of kernel routines with quasi-invariants. Our toolkit includes a tool that automatically generates specialized code by partially evaluating code that refers to quasi-invariants, thus greatly reducing the burden on developers.

Third, since specialized code *assumes* the invariance of quasi-invariant expressions, if a system condition causes an invariant to no longer hold, the corresponding specialized code will produce incorrect results. A correct specialized system must detect quasi-invariant violations, and dynamically recover from them. We refer to the detection of violated quasi-invariants as “guarding.” When a guard indicates that a quasi-invariant has been violated, recovery consists of removing the specialized routine, and “replugging” a less specialized routine that does not assume the invariance of the violated quasi-invariant.

Previous specialization experiments required that programmers manually identify all of the locations within the kernel that may violate quasi-invariants and insert the appropriate guards. Further, once a quasi-invariant has been violated, developers had to manually recover by replugging. We have developed a tool that automatically identifies most of the sites where quasi-invariants may be violated and inserts guards at those locations. To catch the cases our tool may miss (because of C’s lack of type-safety), we have developed another tool to dynamically verify the invariance of a specialized quasi-invariant.

2.3 Other Approaches to Customization

Customizing operating systems is an active area of research. The toolkit approach to specialization distinguishes our work from previous specialization work [26], as well as other customizable operating system projects such as SPIN [3]. The guarding tools we provide support the explicit description and representation of quasi-invariants, thus helping the system preserve correctness despite evolving customization and specialization.

In contrast, systems like SPIN enforce protection through the use of a type-safe programming language combined with a *dispatcher* which enforces constraints described by the service-writer [24]. For example, the dispatcher might enforce 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 damage caused by broken specialized modules to only those tasks that specifically *ask* to use the specialized components. The responsibility of ensuring that specializations do not conflict with each other is left to to extension-writers and the authors of built-in services.

Exokernel [12] represents another approach to operating system customization. Exokernel pushes system services outside the kernel. Exokernel also enforces mainly syntactic protection without explicit description and representation of quasi-invariants. Consequently, the responsibility of ensuring that code fragments outside the kernel 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 flexible replacement of operating system components, particularly *nesting* of operating system components [13, 14] using concepts such as recursive virtual machines [27]. These flexible layers of indirection come at some cost. However, specialization may be able to minimize these costs. The replaceable software components are large and complex, and the relationship between them is largely quasi-invariant, because the components are not replaced frequently. As we will show in Section 4, quasi-invariant relationships between entities in an operating system can be specialized to improve performance.

To summarize our relation to other work in this area, the tools described in this paper build upon previous specialization projects and could be used in conjunction with extensible kernels, such as SPIN, and user-level service based systems, such as Exokernel. In these systems the toolkit would be used to make the assumptions and interdependencies of system extensions explicit, and automatically generate and guard specialized code modules.

3 A Toolkit for OS Specialization

This section describes a toolkit we are developing that provides the assistance programmers need in order to make specialization a usable optimization technique. We begin by reviewing the specialization process, noting where the tools are to be applied. This is followed by a detailed description of each tool.

To aid discussion, we present here a sample quasi-invariant that was used to specialize Linux signal delivery, as described in Section 4.1. The following quasi-invariant asserts that the process sending a signal has the same UID as the target process, and thus has permission to signal the target:

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

The variables `current` and `p` are of type `struct task_struct *`, where `current` is the executing process, and `p` is the target process. The above expression is a quasi-invariant, and the data stored in `current->uid` and `p->uid` are *quasi-invariant terms*.

Postulating Quasi-Invariants As described in Section 1, discovering appropriate conditions to use as quasi-invariants is difficult. The general approach is to use the kernel developer's intuition to postulate that some condition is both quasi-invariant (doesn't change rapidly) and useful (the condition is tested frequently). To answer these questions, the system developer must know all the places in the system where the terms of the quasi-invariant are read and written. The guarding tools described in sections 3.3 and 3.2 automate the process of locating these components, and can assist in determining how frequently a quasi-invariant changes. In our example, the kernel developer would have to determine how frequently the `task_struct.uid` field changes by first locating all the places in the kernel that write to the `task_struct.uid` field, and then profiling the kernel to determine how frequently they occur.

Using Quasi-Invariants to Generate Specialized Code Given some invariant conditions, specialized code can be generated for certain system components. While this can and has been done by hand, it can be automated using *partial evaluation* techniques [7, 29]. Partial evaluation is specifically the idea of defining some of the input to a function to be constant (truly invariant) and using that invariance to optimize the code. Section 3.1 describes our partial evaluation compiler for C code. In our example, the partial evaluation compiler can remove the tests on the `uid` field from the signal delivery code.

Guarding Quasi-Invariants The distinguishing trait of *quasi*-invariants is that they aren't really invariant. Specialized code that depends on quasi-invariants not changing will *break* when the invariants do change. To ensure correctness, the kernel developer must locate all the places in the system that can cause quasi-invariants to change, and *guard* them with code that will *re-specialize* the specialized components to reflect the new state of the quasi-invariants. The tools described in sections 3.3 and 3.2, while useful for postulating quasi-invariants, were primarily designed to assist in placing guards to ensure the applicability of specialized code. In our example, the kernel developer would have to find all places in the kernel that change the UID of a process, and guard them to ensure that they do not break some specialized code.

Replacing Specialized Code Specialized code that depends on quasi-invariants must be re-specialized when the quasi-invariants change. However, the specialized code may be *in use* when the quasi-invariant changes. Therefore, some form of concurrency control must be applied to the quasi-invariants and the specialized code. Section 3.5 describes our tools for efficiently allowing concurrent execution nd replacement of specialized code.

The remainder of this section describes each of our tools.

3.1 Tempo: Generating Specialized Code

Tempo is a program specializer based on partial evaluation [7, 18]. Tempo takes a generic source program P_{gen} written in C plus a known subset of its input (the quasi-invariants), and produces a specialized C program P_{spec}, which is simplified with respect to the quasi-invariants. Tempo supports both compile-time and run-time program specialization [8], but in the specialization experiments carried out so far, we have used only compile-time specialization.

Conceptually, program specialization using partial evaluation is straightforward. Tempo uses the known subset of input to analyze P_{gen}, dividing it into *static* and *dynamic* parts. Immediately, the static part of P_{gen} is evaluated and reduced using the quasi-invariants (the known subset of input), while the dynamic part is copied to the output. The result P_{spec} is usually simpler than P_{gen} since the static part has been pre-computed and only the dynamic part will be executed at run-time. Informally, partial evaluation can be described as an automated propagation of values known to be constant at run-time (typically after some initialization code).

Partitioning program components into static and dynamic parts turned out to be insufficient for C programs in operating systems. To address the complications in operating systems code, several refinements were introduced in Tempo:

- *Static & dynamic* variables: those with value known at specialization time, so they can be exploited in specialization and some code are reduced; but nevertheless some other code is forced to appear in P_{spec}, e.g., due to values of pointers that are difficult to guard.

- *Partially-static structures*: data structures that contain some fields with known values, and other fields that are dynamic.

- *Pointers to partially static structures*: For pointers to partially-static data structures, Tempo must distinguish the static subcomponents from the dynamic ones.

Tempo was used in all of the experiments reported in Section 4 to generate specialized code.

3.2 TypeGuard: Dynamically Guarding Quasi-invariants

TypeGuard is a tool for locating statements in the source code of a program that write to quasi-invariant terms using static type analysis. If the quasi-invariant is a statically-allocated (i.e., global) variable, then guarding the assumption that this property does not change is simple. We can easily locate all program statements that assign to the variable's static name.

Unfortunately, most of the state properties in an operating system that are likely to be quasi-invariant are fields in heap allocated structures. For example, the quasi-invariant `current->uid == p->uid` refers to two *specific* instances of `task_struct`, but there may be hundreds of `task_struct` structs in a running kernel. Finding and guarding all places in the kernel that change state properties on which specialized components depend is the *guarding problem*.

We solve the guarding problem using a combination of static and dynamic methods. Static type checking can locate all kernel source program statements that refer to the struct type and field name that we are concerned with. A *guard* is then placed at each such write. Guards do the run-time checking to decide if the struct being modified is an *instance* of the struct that needs to be guarded, and invoke re-specialization if necessary.

Section 3.2.1 describes our tool to locate updates to variables that require guarding as indicated by type information. Subsection 3.2.2 describes our guards: an efficient run-time method for distinguishing among updates to instances of structures of the guarded type: only those instances pertaining to specialized code require re-specialization when they are updated. Subsection 3.2.3 describes our prototype implementation of *TypeGuard*; a tool for placing the guards described here.

3.2.1 Where to Place Guards

Consider the quasi-invariant

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

The quasi-invariant expression refers to the `uid` field of an `task_struct` structure. Guarding all writes to the `uid` field of the `task_struct` structure is problematic, because there are many instances of the `task_struct` struct. However, we can at least use type checking to locate all of the accesses to structs of *type* `task_struct`.

This method of locating updates to pertinent types is only as effective as the type-safety of the kernel source program. However, we can warn the programmer of type-unsafe operations that may prevent effective location of all statements that need to be guarded. Such operations include:

- type-casted assignment from or to the type of struct with which we are concerned
- attempting to guard a field that is part of a union
- taking the address of a *scalar* field that must be guarded

To explain the last item, consider that the `uid` field is an integer. If a program statement does the following:

```
int * foo = &(current->uid);
```

Then `foo` constitutes a *capability* to violate our quasi-invariant expression. However, we cannot guard all operations using `foo`, because its type is far too generic (most of the system contains `int *` types) and its value may be anonymously passed to other parts of the system. Thus we resort to simply flagging the statement that takes the address of our quasi-invariant term `current->uid`.

3.2.2 Guards: Re-Specialize If Necessary

The method described in Section 3.2.1 suffices to locate all assignments to state variables of the *type* that appear in our quasi-invariant expression, but cannot distinguish among different *instances* of that type. Our specialization concerns only two processes described by two particular `task_struct` structures, yet there are often hundreds of instances of `task_struct` structures in the running kernel.

Whether a quasi-invariant is true of a particular `task_struct` structure is a dynamic property, and so we resort to run-time testing of the quasi-invariant expression to determine whether the update has violated the quasi-invariant. However, it is only a violation of the quasi-invariant if the structure was in fact the one referred to by the expression; the other instances are irrelevant. Furthermore, the quasi-invariant expression may involve several structs, and so testing the expression requires identifying the appropriate instances.

We address these problems by annotating all structs that contain quasi-invariant terms with a special *QUasi-invariant IDentifier* pointer field (QUID). In the case that the `task_struct` struct is the instance referred to in the quasi-invariant expression, the QUID field points to an object that encodes the quasi-invariant expression in such a way that it can perform a *guarded write* to the struct. For example, consider this update to current->uid:

```
    current->uid = bar;
```

A guarded update of the current->uid would be written as:

```
    if (current.QUID != NULL)
        current.QUID->write_uid(bar);
    else
        current->uid = bar;
```

The update_uid function writes the current->uid field in any case, but also atomically adjusts any specialized components that depend on quasi-invariant expressions that depend on this task_struct.uid value.

This guarding code has the property that it very quickly identifies struct instances that are *not* specialized, and dispenses with further checking. However, if the struct does contain quasi-invariant terms, it efficiently locates the code necessary to evaluate the continued validity of the quasi-invariant and invokes the checking code. The guarding code is sufficiently simple that it can be packaged inside a macro, so that the kernel programmer hardly need know it's there:

```
    GW_uid(current, bar);
```

3.2.3 TypeGuard Prototype

We have constructed a tool called *TypeGuard* that does the type-checking described above. The tool is based on the SUIF compiler toolkit [36]. SUIF provides a basic framework that parses C source code and stores it in a standardized intermediate format that is used by each phase of the compiler. We have used the SUIF library of functions for manipulating this intermediate representation to process the program as follows:

1. Locate all declared variables of the type we are concerned with and build a list of their names.

2. Locate all writes to variables of those names.

3. Eliminate assignments to fields that we are not concerned with.

4. Report the remaining statements as locations in the program that need to be guarded.

TypeGuard currently only emits a list of locations in the program that need to be guarded; it does not yet automatically generate the guarding code. Future development of TypeGuard will include a guard generator that actually inserts the guard code rather than just indicating where the guard code should be inserted. Automatic insertion of guard code is relatively simple, but efficient insertion of guard code requires some optimization, such as only checking the quasi-invariant only once after a batch of adjacent updates to quasi-invariant terms.

3.3 Runtime Guarding Through Virtual Memory Protection

The guard locating technique described in Section 3.2 is effective in most cases. However, it is critically dependent on the type-safety of the of the kernel source code and the compiler. Our specialization techniques are aimed at real legacy operating systems, thus we need another guard-placement technique to locate kernel statements that may escape TypeGuard's notice through one of C's many type checking holes.

The *MemGuard* tool is a library of functions that use virtual memory page protection to locate additional kernel operations that require guarding. The basic notion is to set the virtual memory page protection bits of a page containing a quasi-invariant term to *read-only*, in order to trap attempts to change the quasi-invariant value and report them as errors. Writes to other values on the same page that are not recorded as quasi-invariant terms are simply written by the MemGuard trap handler without generating a trap.

Clearly the performance penalty for writing to a page via a trap handler is too high for MemGuard to participate in a performance-oriented system, and that is not MemGuard's purpose. Rather, MemGuard is provided as a *debugging* and development tool for the specialization kernel programmer. The kernel programmer enables MemGuard's protection capabilities, runs the system through a test suite, and then examines the log of quasi-invariant violations produced by MemGuard. The kernel programmer then inserts guards at the locations indicated by MemGuard, and iterates the process until the kernel passes the test suite without complaint from MemGuard.

Figure 1 shows the API provided by the MemGuard library. Section 3.3.1 describes some of the implementation details of MemGuard. Section 3.4.1 describes usage of the MemGuard library.

3.3.1 MemGuard Implementation

When MemGuard is asked to guard a quasi-invariant term, the first task is to turn off write-permissions for that page in virtual memory. In some architecture/OS combinations, such as the HP-UX operating system on the HP-PA architecture [6], we were able to locate protection bits in the virtual memory hardware that were not used by the operating system. However, this is not always possible, and so MemGuard must allocate a page descriptor for each page containing quasi-invariant terms; the page descriptor records whether that page was writable or not *apart* from the protection imposed by MemGuard.

`Enable()/Disable()` Enable and disable the MemGuard facility. This is useful to focus debugging on a particular section of the kernel.

`Protect(addr, size)` Make the data at virtual address `addr` of size `size` a quasi-invariant term, i.e. complain if it is written to.

`Release(addr)` Remove MemGuard protection from the quasi-invariant term at virtual address `addr`.

`Write(addr, size)` Write a new value into the quasi-invariant term at address `addr` of size `size`. It is an error if `size` does not match the size of the quasi-invariant term when it was originally protected. `Write` is used to perform the guarded writes required by TypeGuard. `Write` changes the quasi-invariant term's value without complaining.

Figure 1: The MemGuard API

Once the page is protected, *all* writes to that page will trap to the MemGuard trap handler. Only a small fraction of the writes will be to quasi-invariant terms, the others will be to various kernel data structures that happen to share the page.[1] Thus the page descriptor must also contain a list of the quasi-invariant terms on the page so that writes to the page can be differentiated between writes to quasi-invariant terms (which log error messages) and writes to neighboring kernel data structures (which proceed normally).

The cost of writing to non-protected data structures residing on protected pages is, in part, determined by how quickly the MemGuard trap handler can distinguish between quasi-invariant terms and the neighboring data structures: much of MemGuard's overhead results from this kind of false sharing.

Some regions of memory, such as the kernel's statically allocated data structures, are sufficiently dense that the page descriptors can be laid out in a linear array indexed by the address of the faulting page. This approach is not practical in general, because a 32-bit address space with 8 KB pages results in 524,288 pages. 64-bit machines, such as the HP-PA [6] and the DEC Alpha [30] further aggravate this problem. Thus MemGuard must resort either to hierarchical data structures [28], or explicitly allocated and managed memory regions [31] that are sub-indexed linearly.

Operation	Min	Avg.	Max
Normal write	0	2	1.9
MemGuard write	1624	1971	2434

Table 1: Overhead of MemGuard Writes, in machine cycles

3.4 MemGuard Performance

The time in cycles required for MemGuard to perform various operations is compared to the time for a normal write is presented in Table 1. These measurements were performed on a 100 MHz Pentium PC. The actual overhead of running an operating system with MemGuard active depends on the particular terms being guarded and the workload being measured. However, this overhead should only be incurred by kernel developers during guard-placement trials.

3.4.1 Using MemGuard

As mentioned above, MemGuard is intended to act primarily as a debugging tool to the specialization kernel programmer. The kernel programmer finds undetected updates to quasi-invariant terms by exercising the kernel with a test suite and using MemGuard to detect the writes to quasi-invariant terms. The thoroughness of the guard coverage is a function of the degree to which the test suite exercises the kernel. Access to the OS vendor's test suite enhances MemGuard's utility.

3.5 Replugger: Dynamic Re-Specialization

When a quasi-invariant expression is violated, then the system must adapt itself to the new circumstance without relying on the quasi-invariant. When a guard detects that a quasi-invariant has been violated, it invokes a specialized version-management component, described in [10]. The most common action to be taken by the version-manager is to replace the dependent specialized components with other, differently specialized components, or with generic components. This replacement is called *replugging*, and requires fast, safe, concurrent dynamic linking.

The challenge is to facilitate very low latency execution of a function via an indirect function pointer while concurrently allowing the pointer to be changed. Locks are a logical choice, but locks may substantially degrade performance. In [9], we describe a portable algorithm that supports low-latency invocation of replaceable functions while allowing concurrent update of pointers to those functions.

The need for sophisticated replugging is a function of the kernel and the hardware. Table 2 shows the kinds of

[1] Unfortunately, it is not possible to *move* a quasi-invariant term to a separate page without inducing consistency checking problems at least as difficult as the guarding problem itself.

	Hardware	
Kernel	Uniprocessor	Multiprocessor
Single-threaded	function pointer	simple replugger
Multi-threaded	simple replugger	counting replugger

Table 2: Replugging Needs as a Function of Kernel and Processor Architectures

replugging systems needed for various combinations. The definition of the terms is as follows:

uniprocessor, multiprocessor One CPU vs. multiple CPU's in a shared-memory multiprocessor.

single-threaded, multi-threaded : Whether or not more than one thread from a single process can be concurrently executing a system call, an important distinction when system calls can block inside the kernel.

function pointer No replugging techniques required, since the kernel is a giant mutex over the whole machine. Thus a function pointer suffices, without additional concurrency control requirements.

simple replugger An asymmetric concurrency control mechanism, as described in [9]. This concurrency control mechanism allows fast invocation of the replaceable function, and somewhat slower replacement of the function.

counting replugger An enhancement to the simple replugger that counts the number of threads executing the specialized code. Making such a counting replugger concurrent-safe requires an atomic increment of some kind.

The experiments described in [26] were performed using HP-UX 9.04 (which is single-threaded) on an HP9000 S800 dual processor, and thus we implemented the simple replugger. The experiments described in section 4 of this paper were performed using Linux 2.0 on various single-processor Pentium PCs, and thus only simple function pointers were used.

4 Experiments

This section presents the results of some experiments that evaluate the effectiveness of our specialization toolkit in broad areas of operating system specialization.

We would have preferred to be able to perform a direct comparison between previous hand-specialized results and our tool-assisted specialization of the same systems. Unfortunately, this was not possible for a number of reasons. First, our specialization tools require the system code to be written in ANSI C, but the subject of the previous specialization experiments, HP/UX, is written in K & R C. This was a major factor in our choice of the Linux system for evaluating the toolkit. Second, we could have repeated the hand-specialization experiments on Linux, and compared them to the same specializations done via the toolkit. Unfortunately, the portions of the system that were addressed in previous experiments have already been hand-specialized in the Linux system. Comments from the developers who did this specialization [17] bear out our claim that doing such specialization by hand is difficult and obfuscates the system.

The rest of this section describe our experiences with using the tools to specialize three disparate system components: kernel signal delivery, RPC and memory allocation.

4.1 Signal Delivery

A common source for specialization opportunities comes from transient “connections” between entities in the system. These connections can be explicit, as when a process opens a file to build a connection between the process and the file, or they can be implicit, as when a group of processes repeatedly communicate to achieve some common goal. In any case, the transient connection between entities produces quasi-invariants describing the relationship between the entities.

In previous file system specialization experiments [20, 26], the connection was between a process and a file, and the quasi-invariants related to things such as the location of the file, and whether the file was shared. In this experiment, the connection is between two processes A and B, where A is repeatedly sending UNIX signals to B. We treat the target and destination processes and their relevant properties as quasi-invariant, and specialize the signal delivery mechanism accordingly, using the Tempo partial evaluation compiler augmented with some human assistance.

4.1.1 Specializing Signal Delivery

Figure 2 shows the structure of the `kill` system call. Each of the functions `sys_kill`, `kill_proc`, `send_sig`, and `generate` do some error checking and interpretation on the signal and whether the sender has the right to send it to the target. In addition, `kill_proc` searches the process table for the process with the specified `pid`. The source code for these functions is shown in Figure 3.

Specialization proceeded in two steps. The first was

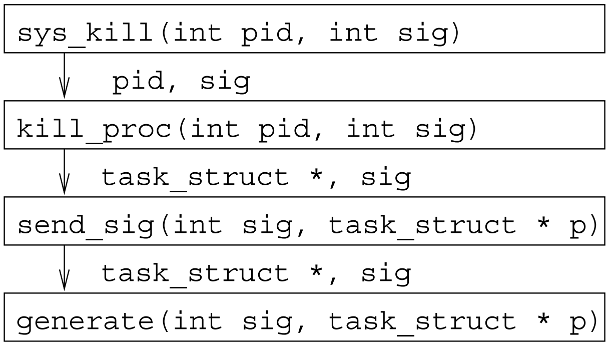
sys_kill(int pid, int sig)
pid, sig
kill_proc(int pid, int sig)
task_struct *, sig
send_sig(int sig, task_struct * p)
task_struct *, sig
generate(int sig, task_struct * p)

Figure 2: Linux kill System Call Architecture

to introduce a caching mechanism that would record the previous target process and signal, and re-use this pointer if a new kill system call is invoked with the same target pid and sig values. The caching mechanism consisted of a task_struct * last_sig_to field in the task_struct structure, and an if statement in kill_proc right ahead of the search of the process table.

The caching mechanism was introduced by hand. While partial evaluation is a powerful technique, it cannot invent data structures and algorithms. In principle, the process table search could be partially evaluated with respect to the pid being searched for. Unfortunately, the current state of partial evaluation technology would require constructing a static process table to search, which is a long and difficult process. The caching mechanism took about 10 minutes to write.

The second step was partial evaluation of the code. The generic kill source code shown in Figure 3 from kill_proc onward was compiled by Tempo, along with the specification that the following values and fields are static:

- the sig and pid parameters
- the current pointer, which points to the currently executing process
- the last_sig and last_sig_to cache fields in the task_struct structure
- the task_struct.pid, task_struct.session, task_struct.euid and task_struct.uid fields in the task_struct structure

The result is a specialized kill_proc procedure called kp_usr1, as shown in Figure 4. Tempo has reduced 90 lines in three functions to a single function of 40 lines, largely by eliminating redundant tests on quasi-invariant properties such as the euid value of the target and destination processes.

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

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

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);
}

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->euid) && (current->euid ^ p->uid) &&
      (current->uid ^ p->euid) && (current->uid ^ p->uid) &&
      !suser ())
    return -EPERM;
  if (!sig)
    return 0;
  /* Forget it if the process is already zombie'd.  */
  if (!p->sig)
    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;
}

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);
}
```

Figure 3: Linux kill System Call Source Code

4.1.2 Signal Specialization Performance

The basic performance impact of signal specialization is shown in Table 3, which compares the latency of sending SIGUSR1 from a process to itself. Time is reported in μ-

```
int kp_usr1 ()
{
  struct task_struct *p;

  {
    int suif_tmp12send_sig_2_1;
    int suif_tmp11send_sig_2_1;

    suif_tmp11send_sig_2_1 = 0;
    suif_tmp12send_sig_2_1 = suif_tmp11send_sig_2_1;
    if (suif_tmp12send_sig_2_1) {
      send_sig_SSSDDDStr_sigaction_DDDSSSS_flat2 = -1;
      goto pprocfin0;
    }
    if (((*(*current).last_sig_to).sig != (void *) 0) == 0) {
      send_sig_SSSDDDStr_sigaction_DDDSSSS_flat2 = 0;
      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;
}
```

Figure 4: Specialized `kill` System Call Source Code: `kill_proc`, `send_sig`, and `generate` Folded and Specialized

Kernel	Latency (μ-seconds)
Standard	44.1
Specialized	15.3

Table 3: Signal Latency: Speedup Due to Specialization

seconds, averaged over four executions of a program that does 100,000 signals. The cost of signalling is a function of the number of processes in the system: in this case, one user was logged in, running an X11 server and three xterm programs and associated shells, and a few other X11 applications running, for a total of 62 processes. Under these (arguably typical) conditions, specialization improved the total latency of signal delivery by 2.87, or 187%.

Total signal latency is composed of several factors, and we did additional experiments to separate those factors out. All of our specialization occurred in the `kill` system call implementation, and did not affect the scheduler, or the signal handler invocation mechanism. Sending a signal of 0 has the semantics of not invoking any signal handler at all, and so we measured the total latency of sending a signal of 0 to isolate the impact on the `kill` system call, as shown in Table 4. Specialization has improved the speed of the `kill` system call by 14.75, or 1,375%. A system call that used to involve approximately 30 μ-seconds of work has been reduced to the null system call.

Kernel	Latency (μ-seconds)
Standard	29.5
Specialized	2.0

Table 4: Signal 0 Latency (no handler invocation): Speedup Due to Specialization

Kernel	Latency (μ-seconds)
Standard	16.7
Specialized	13.8

Table 5: Single User Mode Signal Latency: Speedup Due to Specialization

The work in the `kill` system call consists of searching the process table, and interpreting the state of the parameters and the processes to detect errors and special cases. To minimize the time spent searching the process table, and to minimize other noise from these experiments, we re-ran these experiments in single-user mode, as shown in Table 5. The size of the process table clearly has a major impact on the cost of the `kill` system call, but specialization has still improved performance by 1.21, or 21%.

4.2 Application-level Impact

Finally, one might wonder what kind of application would actually *care* about the performance of repeated signals. Our original intuition was that signals are used between processes to communicate asynchronous information, such as the arrival of a video frame in a buffer [5].

However, this technique is also used to implement more general services: Xavier Leroy's POSIX Threads implementation for Linux [19] uses signals extensively for inter-thread synchronization. Linux threads are somewhat unusual, in that they use kernel level processes with shared address spaces, rather than threads within a single process, much like Plan 9 [25] variable-weight processes. Signals are used to communicate between these processes in the thread library.

Table 6 shows the impact of signal specialization on a test program using this thread library. 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. Table 6 shows that not only has signal specialization improved average performance by 2.11 or 111%, but it has also substantially reduced the variance in the program run time. Single-user mode tests of this program eliminate the variance in both programs, and yield a

	Experiment Run time in seconds				
Kernel	1	2	3	4	Avg
Standard	8.26	13.82	9.35	16.05	11.87
Specialized	6.55	5.07	5.78	5.00	5.60

Table 6: Thread Mutex Performance for Four Runs

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

Figure 5: ANSI-C Malloc interface

```
Vmalloc_t* vmopen(Vmdisc_t* disc,
                  Vmethod_t* meth, int flags);
int vmclose(Vmalloc_t*);
int vmclear(Vmalloc_t*);
int vmcompact(Vmalloc_t* region);
int vmset(Vmalloc_t* region, int flags,
          int type);
Void_t* vmalloc(Vmalloc_t* region,
                size_t size);
Void_t* vmalign(Vmalloc_t* region,
                size_t size, size_t align);
Void_t* vmresize(Vmalloc_t* region, Void_t* addr,
                 size_t size, int type);
int vmfree(Vmalloc_t* region, Void_t* addr);
```

Figure 6: Vmalloc interface

3% benefit for specialization. The small performance gain under these extreme circumstances is expected, but further study is required to find the source of the variance in performance in the multi-user mode.

4.3 Memory Allocation

Dynamic memory allocation is another example where generalized system facilities fail to capitalize on regularity present in real program behaviors. The standard malloc interface is shown in Figure 5. While generic and concise, this interface hides from allocator implementations details that can be used to improve performance.

There are several dimensions to memory allocator performance: latency, fragmentation, and locality. The need to dynamically allocate memory is virtually ubiquitous, so the latency of allocator operations should be minimized. Programmers are quick to code around the allocator if they perceive latency to be unacceptable.

Fragmentation is a measure of how efficient an allocator utilizes of memory. Fragmentation and allocator latency necessarily trade-off against each other. Many studies emphasize that maximizing the allocator's capacity to avoid fragmentation should be the primary design objective [35].

Latency and fragmentation have traditionally been identified as the key performance dimensions for malloc, while locality effects are often overlooked. There are reasons to believe that existing allocators have design traits that cause them to negatively affect the client program's locality of reference.

Internally, allocators commonly employ performance enhancing heuristics, such as boundary tags, that disregard the penalty of polluting the data cache [16]. A boundary tag is a technique that reduces the space overhead of the allocator by placing bookkeeping data inside currently unused memory blocks. This can produce near pathological reference patterns when the allocator does bookkeeping operations.

As the memory allocator has control over where heap data is placed, it follows that the memory allocator has direct influence on the memory reference pattern generated by the client program. Poor placement policy can cause the client to incur more memory penalties than otherwise necessary. Recent work in memory allocation suggests that specializing allocators to real program behavior is imperative in addressing all dimensions of allocator performance [4, 2, 15, 32, 35].

4.3.1 Vmalloc: Towards Specialized Allocation

Vmalloc is an allocator that extends the standard malloc interface with the notion of memory *regions*, each of which has an associated *discipline* for obtaining new memory and a *method* for managing it [32]. Figure 6 contains a portion of the Vmalloc interface. By providing various different disciplines and methods, Vmalloc allows application programmers an ability to tailor memory allocation to their needs.

Many of the specialization strategies proposed for malloc can be mimicked easily with Vmalloc because of the flexibility of its interface. Vmalloc's general purpose allocator is based on a best-fit method which combines use of a splay tree data structure and several performance improving heuristics. Performance of Vmalloc's best-fit allocator is competitive with the best of several popular malloc implementations [32].

Vmalloc provides a transition path to specializing memory allocation of legacy programs. A set of stubs is provided that allows the standard malloc calls to be redirected to Vmalloc. Once Vmalloc is in place, the program can be migrated in pieces to make use of Vmalloc's more specialized methods.

4.3.2 Specializing Opportunities in Vmalloc

Our initial decision to investigate memory allocation as a specialization candidate was that the size argument to malloc is very often static; calls of the form `malloc(sizeof(...))` are commonplace. Our objective is for Tempo to specialize Vmalloc's best-fit allocator based on the quasi-invariant resulting from a static size.

The best-fit allocator contains three distinct strategies corresponding to *tiny*, *small*, and *large* objects respectively. Small objects are the simplest case. They can be handled by indexing to one of a fixed number of linked lists. Tiny objects are handled in a similar way, but require additional work because bookkeeping data that Vmalloc normally stores in the free objects will not fit. Large objects use a linked list to implement some caching of recently freed objects, but falls back to a splay tree when necessary. The splay tree is very effective for dealing with bad allocation patterns, but nevertheless imposes much more overhead in cases where a link list would do. Vmalloc employs several heuristics aimed precisely at avoiding the splay tree. By specializing for static size arguments, we can remove the initial interpretation and directly execute the appropriate strategy. For the small objects, this leads to code similar in spirit of the synthesized allocators produced by CustoMalloc [15].

The region abstraction provided by Vmalloc is yet another example of binding a “connection” between system entities. This binding allows Vmalloc to provide specialized services, but it imposes the familiar interpretation overhead and associated indirections through region data structures. The number of actual regions used by a program would start at one and progress to some relatively small number.

The region parameter given to each Vmalloc operation is dynamic, but once determined it is likely invariant for the lifetime of the program. For our experiment, we concentrate on the core operation of the general purpose allocator, the `bestalloc()` operation provided by the best-fit method. We specialize `bestalloc()` with respect to the default region; that is, the region associated with the standard malloc stubs. The goal is to convert all calls to malloc into calls to this specialized version of `bestalloc`.

By identifying the region and size parameters as quasi-invariant, we use Tempo remove a great deal of the latency in the Vmalloc `bestalloc` operation. We note that, as with caching introduced in the signal experiment, techniques for improving the other performance dimensions of memory allocation require higher level algorithmic specializations than partial evaluation provides. Vmalloc provides a framework for addressing the other dimensions, while the application of Tempo to Vmalloc removes much of the additional latency that would otherwise accompany Vmalloc's flexibility.

4.3.3 Vmalloc Specialization Performance

For our experiment we use a set of benchmark applications provided by Benjamin Zorn's memory allocation repository [37]. The benchmarks are run to measure four scenarios.

The first scenario, LIBC, uses standard malloc implementation provided in the Linux libc library. The second, VMALLOC, uses the unmodified Vmalloc via the malloc compatibility stubs. The remaining scenarios, which we call SynthoVmalloc1 and SynthoVmalloc2, measure two alternative approaches to deploying Tempo specialized Vmalloc. The two specialized Vmallocs are distinguished by whether the original programs are recompiled.

In SynthoVmalloc1, we are interested in measuring the feasibility of linking against a shared library version of SynthoVmalloc. Invoking a function in a shared library has overhead, but saves system space. The overhead will negate some of the advantages gained through specialization but we have still converted multiple interpretations into one up-front dispatch. This can be thought as a time-space tradeoff where interpretation time has been traded for cost of replication of code space. If code blowup becomes problematic, this technique might prove useful.

SynthoVmalloc2 aims for maximum specialization. Client code is recompiled, so that fully static invocations can make use of inlining.

```
 Note to program committee:  a bug
in one of our tools prevents us
from correctly parsing the vmalloc
source code.  This bug is unrelated to
performance, and thus will not affect
the results reported elsewhere in
this paper.  We are very confident
that we can fix this bug by the
final paper deadline, and expect
performance results similar to our
other experiments.
```

4.3.4 Further Experiments

We are interested in pursuing the locality dimension of allocator performance, but we need new profiling tools. Current profiling tools do not measure locality effects. We plan to make use of Pentium hardware features which allow measurement of several kinds of cache event. The same specializations above on size and region will be used in conjunction with the specialized methods of Vmalloc to improve locality. For example, we can identify important common large object sizes and allocate them from using Vmalloc's pool method. Aside from decreased memory overhead this affords, [4] shows how common initializations can be avoided to improve locality.

4.4 RPC Specialization

The specialization of Sun RPC (proposed in [34]) was the first successful application of systems code using partial evaluation, in particular the Tempo program specializer [8] (summarized in Section 3.1). Since the RPC experiment is being reported in another submission to SOSP'97 (*Automatic Specialization of Sun RPC Using a Partial Evaluator*, by anonymous authors), we only outline the main results here.

The RPC experiment applied the Tempo program specializer to post process the client stub code produced by Sun `rpcgen` stub generator. Their main idea is to take advantage of the values declared at RPC initialization time that remain constant through the execution of subsequent RPC calls. Examples of these static values include the choice of underlying protocol (e.g., UDP), plus the number and type of RPC parameters. Tempo takes the static parameters, and specializes the client stub, producing performance gains of between 2 to 3.5 times speedup in RPC microbenchmarks and between 13% and 22% speedup for an application program using RPC to send and receive integers of an array.

From the specialization toolkit point of view, the RPC experiment is notable since the Sun RPC is commercial, mature code. Two advantages follow from the application of Tempo to commercial code. First, by preserving the original source code, Tempo preserves the system maintainability and safety for the programmer. Second, the successful use of Tempo to representative commercial system shows the promise of applying Tempo to other industrial strength operating systems code.

5 Related Specialization Tools

This section describes some related work developing tools for specialization. Some of the tools described were actually designed for specialization, such as the C-Mix [1] partial evaluation compiler. Others are general-purpose software engineering tools that just happen to be useful for specialization, such as the Lackwit C analysis tool.

5.1 C-Mix Partial Evaluation Compiler

C-Mix [1] is the only other partial evaluator for C programs besides Tempo. Like Tempo, C-Mix can partially evaluate C programs, do inter-procedural analysis, and deal with complex data structures and side-effects. However, it was not specifically designed to deal with systems code, and thus its analysis is not as precise as Tempo's. In particular, C-Mix is point 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 is also more consumptive of code space, because it eagerly replicates code to avoid problems in binding-time analysis.

5.2 Lackwit C Program Understanding Tool

Lackwit [22] is a program understanding tool for C based on type inference. Lackwit discards C's type system as too weak to be useful, and instead infers its own dynamic types for values based on the set of operations the value participates in, derived from a conservative data flow analysis of the program. Thus Lackwit can construct rather interesting 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 very useful in placing guards for quasi-invariants in system code, similar to TypeGuard. Lackwit performs more precise analysis than TypeGuard, but at the expense of using an algorithm that is NP-hard in the worst case.

5.3 OMOS Dynamic Linking Tool

The Utah Flex project developed OMOS [23], an object/meta-object server that allows the dynamic linking of executable modules. OMOS provides for the dynamic instantiation of executable modules, and wraps them 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 quasi-invariant guards, but does the checking only at load time.

6 Experiences and Discussion

This section summarizes our experiences with tool-based specialization, beyond the performance improvements described in Section 4, and highlights some ideas for future research. The discussion ranges from comments on the status and effectiveness of specific tools to more general statements about the fundamental obstacles to the widespread propagation of tool-based specialization.

6.1 Experiences with Specialization Tools

Not surprisingly, we found out that operating systems C code is not an easy target for partial evaluation tools. We discovered that despite Tempo's state of the art binding time analysis (BTA), we still had to find work arounds to help it optimizing certain code paths containing pointers. For example, it does not currently handle assertions about fields of structs accessed via pointers unless a concrete instance of the struct is provided. Such examples are common in operating system code. Two approaches seem promising in addressing this problem.

First, instead of trying to make BTA more sophisticated in handling the obscure situations caused by pointers, frequently we found it more appealing to improve obscure code. This indicates the desirability of semantically cleaner programming languages, such as Java, ML or Modula-3, for building specialization-friendly operating systems. In particular, the use of Java for implementing JavaOS has already shown some interesting optimization opportunities, such as copy-elimination [21].

A second solution to the problem of analyzing pointers is to make the various stages of compilation explicit in the original code. We have begun investigation into how such staging should be expressed, using an object-oriented paradigm [10, 33]. Other examples related to this kind of approach are the use of an explicit eval function in functional programming and the 'C (tick C) approach [11].

6.2 Experiences with Guarding Tools

The correctness and performance of a system containing specialized code depend on the correctness and performance of the guarding tools. There are interesting correctness and performance trade-offs for each of the guarding tools proposed in this paper.

MemGuard is guaranteed to catch all write accesses to guarded locations. However, this degree of correctness comes at the expense of high overhead for page protection fault and single-step trap handling. Performance could be improved somewhat by reducing the number of unnecessary page protection faults due to false sharing by laying out data such that guarded locations are allocated on their own private pages, or by employing hardware techniques, such as Liedtke's fine-grained page tables. The overhead of single-step trap handling could also be avoided by simulating the completion of the faulting write instruction, but at the expense of significantly increased complexity in the tool.

Instead of guarding accesses to quasi-invariant terms at run time, TypeGuard attempts to identify all writes to fields of structs of a certain type at compile-time. However, types that are frequently used but rarely specialized can impose a more guarding overhead than benefit gained from specialization. TypeGuard's coverage is limited by the type safety of the particular program being analyzed: arithmetic on variables of type `void *` make *all* type analysis irrelevant. In a type-safe language such as Java, the TypeGuard approach will be able to catch all such writes.

Manual specialization is a potential source for introducing bugs into systems. In contrast, tool-based specialization has the potential to *ease* the software complexity problem pointed out by the industry panel at OSDI'96. Code that has been hand-specialized for performance tends to be more complex and difficult to maintain, aggravating the cost of OS development. *Generic* code, which is correct but *not* specialized for various circumstances, is relatively simple and easy to maintain. Automatic program specializers have the potential to transform such generic code into specialized code, combining most of the performance of hand-tuned code with the maintainability of generic code.

6.3 Summary and Future Work

Overall, we observed significant leverage in managing the specialized code. Specialization by hand would have required either multiple versions of the source, or a lot of conditional compilation. With Tempo, the original code remained mostly untouched. For instance, once the initial problems of partially evaluating the signal code were solved, dozens of specialized versions of the `kill` system call implementation followed in just a few minutes. We expect the Tempo-produced specialized code to be much more amenable to later examination than would be the case for a hand specialized version.

The tools presented in this paper aid in the production of specialized code paths and in guarding them. Another important problem is how to identify good opportunities for specialization. In all our experiments to date, we have identified them by hand, using expert knowledge and heuristics to determine whether they would be good opportunities for specialization.

It would be useful to have tools to identify hot spots in operating systems, distill quasi-invariants 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, which specialized versions to generate ahead of time, which ones to cache, and what policies to use for managing such a cache.

Our experience shows the definite usefulness of our first generation specialization tools. We are in the process of developing more tools to increase the degree of automation of the specialization process.

7 Conclusions

Even though specializing operating systems has been demonstrated to have the potential of significant performance improvements, experience with specialization has been limited to only a part of the research community. This paper described a toolkit that should enable specialization to be used by more operating systems developers, both research and commercial. We have evaluated the effectiveness of the toolkit by using the tools to successfully specialize a broader range of operating system components than has previously been possible. The resulting components performed significantly better than their unspecialized versions, as we had hoped.

Furthermore, we found that *automated* specialization combined with tool-assisted guarding provides the added benefit of improving the maintainability of optimized code, by obviating significant changes to the original source. In this regard, successful experiments with *production* operating system code demonstrate the potential value of the toolkit beyond the research community.

Our experience with the toolkit has suggested areas for further work, particularly the creation of tools that assist developers identify useful opportunities for specialization. Based the experiences reported in this paper, we see tool-based specialization emerging as a key development tool for efficient, adaptive and maintainable operating systems.

References

[1] L.O. Andersen. Binding-time Analysis and the Taming of C Pointers. In *Proceedings of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 47–58, Copenhagen, Denmark, June 1993.

[2] David A. Barrett and Bejamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, June 1993.

[3] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[4] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference*, 1994.

[5] Authors anonymized for review. A Distributed Real-Time MPEG Video Audio Player.

[6] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX Operating System on HP Precision Architecture Computers. *Hewlett-Packard Journal*, 37(12):4–22, December 1986.

[7] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[8] Charles Consel and Francois Noël. A general approach to run-time specialization and its application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburgh Beach, FL, January 1996.

[9] Authors anonymized for review. Fast Concurrent Dynamic Linking for an Adaptive Operating System.

[10] Authors anonymized for review. Specialization Classes: An Object Framework for Specialization.

[11] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A Language for High-Level, Efficient, and Machine-Independent Dynamic Code Generation. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburgh Beach, FL, January 1996.

[12] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[13] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–151, October 1996.

[14] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, October 1996.

[15] Dirk Grunwald and Benjamin Zorn. Customalloc: Efficient synthesized memory allocators. *Software—Practice and Experience*, 23(8):851–869, August 1993.

[16] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, June 1993.

[17] Michael K. Johnson. The Linux Kernel Hacker's Guide: A Tour of the Linux VFS. `http://www.redhat.com:8080/HyperNews/get/fs/vfstour.html`, 1996.

[18] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[19] Xavier Leroy. The LinuxThreads library. `http://pauillac.inria.fr/~xleroy/linuxthreads/`, 1996.

[20] Authors anonymized for review. Threads and Input/Output in the Synthesis Kernel.

[21] James G. Mitchell. JavaOS: Back To The Future. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 1, October 1996. Invited talk.

[22] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. In *Proceedings of International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.

[23] Doug Orr. OMOS - an object server for program execution. In *Proc. International Workshop on Object-Oriented Operating Systems*, 1992.

[24] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an Extensible System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–212, October 1996.

[25] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer UKUUG Conference*, pages 1–9, London, UK, July 1990.

[26] Authors anonymized for review. Optimistic Incremental Specialization: Streamlining a Commercial Operating System.

[27] Authors anonymized for review. The Synthesis Kernel.

[28] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6), June 1990.

[29] P. Sestoft and A. V. Zamulin. Annotated bibliography on partial evaluation and mixed computation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[30] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.

[31] Avadis Tevanian, Jr. *Architecture-Independent Virtual Memory Manager for Parallel and Distributed Environments: The Mach Approach*. PhD thesis, Carnegie Mellon University, December 1987.

[32] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience*, 26(3):357–374, March 1996.

[33] Authors anonymized for review. Declarative Specialization of Object-Oriented Programs.

[34] Eugen-Nicolae Volanschi, Gilles Muller, and Charles Consel. Safe Operating system Specialization: The RPC Case Study. In *Proceedings of the First Annual Workshop on Compiler Support for System Software*, Tuscon, AZ, February 1996.

[35] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, LNCS. Springer Verlag, 1195.

[36] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasighe, Jennifer M Anderson, Steve K.K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. `http://suif.stanford.edu/suif/suif-overview/suif.html`.

[37] Benjamin Zorn. A collection of malloc benchmarks. ftp://ftp.cs.colorado.edu/pub/misc/malloc-benchmarks/.