Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem*
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

Abstract

Systems software such as OS kernels, embedded systems, and libraries must obey many rules for both correctness and performance. Common examples include "accesses to variable A must be guarded by lock B," "system calls must check user pointers for validity before using them," and "message handlers should free their buffers as quickly as possible to allow greater parallelism." Unfortunately, adherence to these rules is largely unchecked.

This paper attacks this problem by showing how system implementors can use *meta-level compilation* (MC) to write simple, system-specific compiler extensions that automatically check their code for rule violations. By melding domain-specific knowledge with the automatic machinery of compilers, MC brings the benefits of language-level checking and optimizing to the higher, "meta" level of the systems implemented in these languages. This paper demonstrates the effectiveness of the MC approach by applying it to four complex, real systems: Linux, OpenBSD, the Xok exokernel, and the FLASH machine's embedded software. MC extensions found roughly 500 errors in these systems and led to numerous kernel patches. Most extensions were less than a hundred lines of code and written by implementors who had a limited understanding of the systems checked.

1 Introduction

Systems software must obey many rules such as "check user permissions before modifying kernel data structures," "for speed, enforce mutual exclusion with spin locks rather than disabling interrupts," and "message handlers must free their buffer before completing." Code that does not obey these rules can degrade performance or crash the system.

There are several methods to find violations of system rules. A rigorous way is to build an abstract specification of the code and then use model checkers [23, 32] or theorem provers/checkers [2, 11, 25] to check that the specification is internally consistent. When applicable, formal verification finds errors that are difficult to detect by other means. However, specifications are difficult and costly to construct. Further, specifications do not necessarily mirror the code they abstract and, in practice, suffer from missing features and over-simplifications. While recent work has begun attacking these problems [6, 14], it is extremely rare for software to be verified.

The most common method used to detect rule violations is testing. Testing is simpler than verification. It also avoids the mirroring problems of formal verification by working with actual code rather than an abstraction of it. However, testing is dynamic, which has numerous disadvantages. First, the number of execution paths typically grows exponentially with code size. Thorough, precise testing requires writing many test cases to exercise these paths and drive the system into error states. The effort required to create these tests, and the time it takes to run them, scales with the amount of code. As a result, real systems have many paths that are rarely or never hit by testing and errors that manifest themselves only after days of continuous execution. Further, finding the cause of a test failure can be difficult, especially when the effect is a delayed system crash. Finally, testing requires running the tested code, which can create significant practical problems. For example, testing all device drivers in an OS requires acquiring possibly hundreds or thousands of devices and understanding how to thoroughly exercise them.

Another common method to detect rule violations is manual inspection. This method has the strength that it can consider all semantic levels and adapt to ad hoc coding conventions and system rules. Unfortunately, many systems have millions of lines of code

*This research was supported in part by DARPA contract MDA904-98-C-A933 and by a Terman Fellowship.

with deep, complex code paths. Reasoning about a single path can take minutes or sometimes, when dealing with concurrency, hours. Further, the reliability of manual inspection is erratic.

These methods leave implementors in an unfortunate situation. Verification is impractical for most systems. Testing misses many cases and makes diagnosis difficult. Manual inspection is unreliable and tedious. One possible alternative is to use static compiler analysis to find rule violations. Unlike verification, compilers work with the code itself, removing the need to write and maintain a specification. Unlike testing, static analysis can examine all execution paths for errors, even in code that cannot be conveniently executed. Further, a compiler analysis pass reduces the need to construct numerous test cases and scales from a single function to an entire system with little increase in manual effort.

Compilers can be used to enforce systems rules because many rules have a straightforward mapping to program source. Rule violations can be found by checking when source operations do not make sense at an abstract level. For example, ordering rules such as "interrupts must be enabled after being disabled" reduce to observing the order of function calls or idiomatic sequences of statements (in this case, a call to a disable interrupt function must be followed by a re-enable call).

The main barrier to a compiler checking or optimizing at this level is that while it must have a precise understanding of the semantics of its input code, it typically has no idea of the "meta" semantics of the software system this code constructs. Thus, it cannot check many properties inexpressible (or just not expressed) in terms of the underlying language's type system. This leaves an unfortunate dichotomy. Implementors understand the semantics of the system operations they build and use but do not have the mechanisms to check or exploit these semantics automatically. Compilers have the machinery to do so, but their domain ignorance prevents them from exploiting it.

This paper shows how to automatically check systems rules using *meta-level compilation* (MC). MC attacks this problem by making it easy for implementors to extend compilers with lightweight, system-specific checkers and optimizers. Because these extensions can be written by system implementors themselves, they can take into account the ad hoc (sometimes bizarre) semantics of a system. Because they are compiler based, they also get the benefits of automatic static analysis.

In our MC system, implementors write extensions in a high-level state-machine language, *metal*. These extensions are dynamically linked into our extensible compiler, *xg++*, and applied down all flow paths in all functions in the program source input. They use language-based patterns to recognize operations that they care about. Then, when the input code matches these patterns, they detect rule violations by transitioning between states that allow or disallow other operations.

This paper's primary contribution is its demonstration that MC is a general, effective approach for finding system errors. Our most important results are:

1. MC checkers find serious errors in complex, real systems code. We present a series of extensions that found roughly 500 errors in four systems: the Linux 2.3.99 kernel, OpenBSD, the Xok exokernel [16], and the FLASH machine's embedded cache controller code [20]. Many errors were the worst type of systems bugs: those that crash the system, but only after it has been running continuously for days.

2. MC optimizers discover system-level opportunities that are difficult to find with manual inspection. While the main focus of this paper is error checking, MC extensions can also be used for optimization. Section 8 describes three FLASH-specific, MC optimizers that found hundreds of system-level optimization opportunities.

3. MC extensions are simple. The extensions mentioned above are typically less than a hundred lines of code.

A practical result of our experience with MC is that the majority of our extensions were written by programmers who had only a passing familiarity with the systems that they checked. Although writing code that obeys system rules can be quite difficult, these rules are easy to express. Thus, writing checkers for many of them is relatively straightforward.

This paper is laid out as follows. Section 2 discusses related work. Section 3 gives an overview of MC and the system we use to implement it. Section 4 applies the approach to the C `assert` macro and shows that even in such a limited domain, MC provides non-trivial benefits. Section 5 shows how to use MC to enforce ordering constraints such as checking that kernels verify user pointers before using them. Section 6 extends this to global, system-wide constraints. Section 7 is a more detailed case study in how we used MC to check Linux locking and interrupt disabling/re-enabling disciplines. Section 8 describes our FLASH optimizers, and Section 9 concludes.

2 Related Work

We proposed the initial idea of MC in [9] and provided a simple system, `magik` (based on the `lcc` ANSI C compiler [12]), for using it. While the original paper had many examples, it provided no experimental evaluation. This paper provides a more developed view of MC, a significantly easier-to-use and more powerful framework for building extensions, and an experimental demonstration of its effectiveness. Concurrently with this paper, we presented a detailed case study of applying MC to the FLASH system [4]. The 8 compiler extensions presented in that paper discovered 34 errors in FLASH code that could potentially crash the machine, such as message handlers that lost or double freed hardware message buffers and buffer race conditions. This paper's main difference is its demonstration that MC is a general technique by applying it to a variety of systems. Because of this broader scope, it lacks the detail in [4], but finds roughly a factor of ten more errors.

Below, we compare our work to efforts in high-level compilation, verification, and extensible compilers.

Higher-level compilation. Many projects have hard wired application-level information in compilers. These projects include: compiler-directed management of I/O [24]; the ERASER dynamic race detection checker [30]; ParaSoft's Insure++ [19], which can check for Unix system call errors; the use of static analysis to check for security errors in privileged programs [1]; and the GNU compilers' `-Wall` option, which warns about dangerous functions and questionable programming practices. Related to the checkers in this paper, Microsoft has an internal tool for finding a fixed set of coding violations in Windows device drivers [27] such as errors in handling 64-bit code and missing user pointer validity checks.

These projects use compiler support to analyze specific problems, whereas MC explicitly argues for the general use of compilers to check and optimize systems and provides an extensible framework for doing so. This extensibility enables detection of rule violations that are impossible to find without system-specific knowledge.

Systems for finding software errors. Most approaches to statically finding software errors center around either formal verification (as discussed in Section 1) or strong type checking.

Verification uses stronger analysis than MC extensions. However, MC extensions appear to be more generally effective. To the best of our knowledge, verification papers tend to find a small number of errors (typically 0-2), whereas the MC checkers in this paper found hundreds. Verification's lower bug counts seem largely due to the difficulty in writing specifications, which scales with code size. As a consequence, only small pieces of code are verified. In contrast, because MC operates directly on source code, it (like traditional compiler analyses) applies as easily to millions of lines of code as it does to only a few.

Two recent strong-typing systems are the extended static type checking (ESC) project [8] and Intrinsa's PREfix [15]. Both of these systems use stronger analyses than our approach. However, they only check for a fixed set of low-level errors (e.g., buffer overruns and null pointer references). Their lack of extensibility means that, with the exception of ESC's support for finding some class of race conditions, neither system can find the system-level errors that MC can detect.

LCLint [10] statically checks programmer source annotations to detect coding errors and abstraction barrier violations. Like ESC and Intrinsa, LCLint is not extensible, which prevents it from finding the errors that MC can find. Further, the source annotations that LCLint requires scale with code size, significantly increasing the manual effort needed to apply it.

Extensible compilation. There have been a number of "open compiler" systems that allow programmers to add analysis routines, usually modeled as extensions, that traverse the compiler's abstract syntax tree. These include Lord's `ctool` [22], which allows scheme extensions to walk over an abstract syntax tree for C, and Crew's Prolog-based ASTLOG [7], also used for C.

Lamping et al. [21] and Kiczales et al. [17] argue for pushing domain-specific information into compilation. They use meta-object protocols (MOPs) to allow programs to be augmented with a "meta" part that controls the base [17]. Such protocols are typically dynamic and have fairly limited analysis abilities. Shigeru Chiba's `Open C++` [3] provides a static MOP that allows users to extend the compilation process.

The extensions in these systems are mainly limited to syntax-based tree traversal or transformation and do not have data flow information. As a result, they seem to be both less powerful than MC extensions and more difficult to use. Our current, language-based approach is a dramatic improvement over our previous tree-based systems: extensions are 2-4 times smaller, have less bugs, and handle more cases. Further, to the best of our knowledge, `ctool`, ASTLOG, and `Open C++` provide no experimental results, making it difficult to evaluate their effectiveness.

At a lower-level, the ATOM object code modification system [31] gives users the ability to modify object code in a clean, simple manner. By focusing on machine code, ATOM can be used in more situations than MC, which requires source code. However, while dynamic testing schemes [13, 30] are well served by object-level modifications, it would be difficult to perform our static checks without the semantic information available in the compiler.

Concurrently with our original work [9], Kiczales et al. [18] proposed "aspect oriented programming" (AOP) as a way of combining code that manages "aspects," such as synchronization, with code that needs them. AOP has the advantage of being integrated within a traditional language framework. It has the disadvantage that aspects have more limited scope than MC extensions, which survey the entire system as well as check rules difficult to enforce with an AOP framework (e.g., preventing kernel code from using floating point). Further, because AOP requires source modifications, retro-fitting it on the systems we check would be non-trivial.

3 Meta-level Compilation

Many systems constraints describe legal orderings of operations or specific contexts in which these operations can or cannot occur. Since the actions relevant to these rules are visible in program source, an MC compiler extension can check them by searching for the corresponding operations and verifying that they obey the given ordering and/or contextual restrictions. Table 1 gives a representative set of rule "templates" that can be checked in this manner along with examples. Many system rules that roughly follow these templates can be checked automatically. For example, an MC extension to enforce the contextual rule, "for speed, if a shared variable is not modified, protect it with read locks," can search for each write-lock critical section, examine all variable uses, and, if no stores occur to protected variables, demote the locks or suggest alternative usage.

3.1 Language Overview

In our implementation of MC, compiler extensions are written in a high-level, state-machine language, *metal* [5]. These extensions are dynamically linked into our extensible compiler, *xg++*(based on the GNU g++ compiler). After *xg++* translates each input function into its internal representation, the extensions are applied down every possible execution path in that function. The state machine part of the language can be viewed as syntactically similar to a "yacc" specification. Typically, SMs use patterns

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
             | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
        { err("exiting w/intr disabled!"); }
     ;
}
```

Figure 1: A *metal* SM to detect (1) when interrupts disabled using `cli` are not re-enabled using either `sti` or `restore_flags` and (2) duplicate enable/disable calls.

to search for interesting source code features, which, when matched, cause transitions between states. Patterns are written in an extended version of the base language (C++), and can match almost arbitrary language constructs such as declarations, expressions, and statements. Expressing patterns in the base language makes them both flexible and easy to use, since they closely mirror the source constructs they describe.

Figure 1 shows a stripped-down *metal* extension for Linux that checks that disabled interrupts are re-enabled or restored to their initial state upon exiting a function. Interrupts are disabled by calling the `cli()` procedure; they are enabled by calling `sti()` or restored using `restore_flags(flags)`, where the `flags` variable holds the interrupt state before the `cli()` was issued. Conceptually, the extension finds violations by checking that each call to disable interrupts has a matching enable call on all outgoing paths. As refinements, the extension warns of duplicate calls to these functions or non-sequitur calls (e.g., re-enabling without disabling). A more complete version of this checker, described in Section 7, found 82 errors in Linux code.

The extension tracks the interrupt status using

Rule template	Examples
"Never/always do X"	"Do not use floating point in the kernel." (§ 4.3) "Do not allocate large variables on the 6K byte kernel stack." (§ 4.3) "Do not send more than two messages per virtual network lane." "Allocate as much storage as an object needs." (§ 5.2)
"Do X rather than Y"	"Use memory mapped I/O rather than copying." "Avoid globally disabling interrupts."
"Always do X before/after Y"	"Check user pointers before using them in the kernel." (§ 5.1) "Handle operations that can fail (e.g., memory, disk block, virtual interrupt allocation)." (§ 5.2) "Re-enable interrupts after disabling them." (§ 7) "Release locks after acquiring them." (§ 7) "Check user permissions before modifying kernel data structures."
"Never do X before/after Y"	"Do not acquire lock A before B." "Do not use memory that has been freed." (§ 5.2) "Do not (deallocate an object, acquire/release a lock) twice." (§ 5.2 § 7) "Do not increment a module's reference count after calling a function that can sleep." (§ 6.3)
"In situation X, do (not do) Y"	"Protect all variable mutations with write locks." "If a system call fails, reverse all side-effect operations (deallocate memory, disk blocks, pages, unincrement reference counters)." (§ 5.2 § 6.3) "To avoid deadlock, while interrupts are disabled, do not call functions that can sleep." (§ 6.2)
"In situation X, do Y rather than Z"	"If a variable is not modified, protect it with read locks." "If code does not share data with interrupt handlers, then use spin locks rather than the more expensive interrupt disabling." "To save an instruction when setting a message opcode, xor in the new and old opcode rather than using assignment." (§ 8)

Table 1: Sample system rule templates and examples. Checkers for the rule are denoted by section number.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);
        return bh;
}
```

Figure 2: Example code from the Linux 2.3.99 Raid 5 driver illustrating a real error caught by the extension. The SM will be applied down both paths in this function. The path ending with a return of `bh` is well formed and will be accepted. The path ending with the return of `NULL` is not, and will get a warning about not re-enabling interrupts.

two states, `is_enabled` and `is_disabled`. SMs start in the state mentioned in the first transition definition (here, `is_enabled`). Each state has a set of rules specifying a pattern, an optional state transition, and an optional action. Actions can be arbitrary C++ code. For a given state, *metal* checks pattern rules in lexical order. If any code matches the specified patterns, *metal* processes this matching code, sets the state to the new state (the token after the `==>` operator), and executes the action. In this example, `is_enabled` has two rules. The first, actionless rule searches for functions that disable interrupts using the `disable` pattern and transitions to the `is_disabled` state. The second rule searches for calls to functions that enable interrupts and gives a warning. Since it does not specify a transition state, the SM remains in the `is_enabled` state. If no pattern matches, the SM remains in the same state and continues down the current code path. The `flags` variable is a wild card that matches any expression of type `unsigned`. When it is matched, *metal* will put the matching expression in `flag`, which can then be used in an action. We use this feature in an extension discussed in Section 4.

To run this SM, it is first compiled with `mcc`, our *metal* compiler. It is then dynamically linked into *xg++* using a compile-time, command-line flag. When run on the Linux "RAID 5" driver buffer allocation code in Figure 2, it is pushed down both paths in the function. The first path returns `NULL` when the buffer pool is empty (i.e., when the `if` statement fails); the other returns a buffer on successful allocation. The first path fails to re-enable interrupts, and this error [1] is caught and reported by the extension.

One way to get a feel for how costly it would be to manually perform the check our SM does automatically is that even when we showed an experienced Linux programmer the exact error in Figure 2, it took him over 20 minutes to examine a single call chain out of the nine leading to this function. Performing similar analysis for the other hundreds of thousands of lines of driver and kernel code seems impractical.

3.2 Practical issues

Metal SMs can specify whether they should be applied either down all paths (i.e., flow-sensitive) or linearly through the code (i.e., flow-insensitive). A simple implementation of flow-sensitive SMs could take exponential time in some cases. We use aggressive caching to prune redundant code paths where SM instances follow paths that join (e.g., if statements, loops) and reach the join point in the same state. Our caching is based on the fact that a deterministic SM applied to the same input in the same internal state must compute the same result. The system represents the state of an SM as a vector holding the value of its variables. For each node in the input flow-graph, it records the set of states in which it has been visited. If an SM arrives at a node in the same state as a previous instance, the system prunes it.

While caching was originally motivated by speed, perhaps its most important feature is that it provides a clean framework for computing loop "fixed points" transparently. When an SM has exhausted the set of states reachable within the loop (typically with two iterations), *metal* automatically stops traversing the loop. This fixed-point behavior depends on the SM having a finite (and small) number of states. We do not currently enforce this restriction.

The current *xg++* system does not integrate global analysis with the SM framework. Instead, it provides a library of routines to emit client-annotated flow graphs to a file, which can then be read and traversed. Section 6 gives an example of how we used this framework to compute the transitive closure of all possibly-sleeping functions. We are integrating these two passes.

[1] Amusingly, this interrupt disable bug would be masked by an immediate kernel segmentation fault since callers of this function dereference the returned pointer without checking whether the allocation succeeded.

3.3 Caveats

Most of our extensions are checkers rather than verifiers: they find bugs, but do not guarantee their absence. For example, their ignorance of aliases prevents them from asserting that many actions "cannot happen." In general, many compiler problems are undecidable, which places hard limits on the effectiveness of static analysis. Despite these limitations, as our results show, MC extensions are quite effective. We are currently investigating how to turn some classes of checkers into verifiers.

We mainly check systems we did not build. As a result, some rule violations we found might not be bugs because the code could use a non-obvious system feature that works correctly in a specific situation. We countered this danger in two ways. First, we sent our error logs to the system implementors of Linux, FLASH, and Xok for confirmation. However, while we got feedback on many errors, their sheer number meant that many did not receive careful examination. Second, we conservatively did not count many cases that were difficult to reason about. While our results may still contain mis-diagnoses, we would be surprised if these caused more than a few percentage points difference.

Several of our checkers produce a number of false positives (in the worst case, in Section 7, up to three per error). These are due to the limitations of both static analysis and our checkers, which primarily use simple local analyses. Usually these numbers can be reduced significantly by adding some amount of global analysis or system-specific knowledge. In almost all cases, each false positive can be suppressed with a single source annotation. Extensions can provide annotations by supplying a set of reserved functions that clients call to indicate that a specific source-level warning should be suppressed. As a refinement, checkers can detect bogus or erroneous annotations by warning when they are not needed.

Basing our MC system on a C++ compiler has caused difficulties when applying it to Linux and Xok. These systems aggressively assume C's more relaxed type system and use GNU extensions that are illegal in `g++`. Thus, while in theory MC can be applied to a system transparently, we had to modify Xok and Linux to remove GNU C constructs that are illegal in C++. We also modified the `g++` front-end to relax its type checking. To avoid this labor for other systems, we are currently finishing a `gcc`-based implementation of *xg++*. More generally, since the *metal* language has been designed to be shielded from both the underlying language and compiler, we plan to port it other languages and other compilers.

The remainder of this paper describes the extensions we implemented using *metal* and *xg++* and the results of applying the concept of meta-level compilation to real systems.

4 A Simple Meta-language

The C `assert` macro takes a single condition as its argument, checks this condition at runtime, and aborts execution if the condition is false. This macro defines one of the simplest meta-languages possible: it has no state and a single operation. This section shows how MC can help even such simple interfaces by presenting two extensions that check the following two assertion invariants:

1. Assertions should not have non-debugging side-effects. Frequently, `assert` is used only for development and turned off in production code. If an `assert` condition has important side-effects, these will disappear and the program will behave incorrectly.

2. Assertion conditions should not fail. Programmers use assertions to check for conditions that should not happen. Any code path leading to an assertion that causes its boolean expression to fail is probably a bug.

4.1 Checking assertion side-effects

Figure 3 presents a *metal* checker that inspects assertion expressions for side-effects. The directive, "`flow_insensitive`," tells *metal* to apply the extension linearly over input functions rather than down all paths, improving speed and error reporting (since there will be exactly one message per violation). The SM begins in the initial state, `start`, and uses the literal *metal* pattern "`{assert(expr);}`" to find all `assert` uses. [2] On each match, *metal* stores the `assert` expression in the variable, `expr`. It then runs `start`'s action, which uses the *metal* procedure `mgk_expr_recurse` to recursively apply the SM to the expression in `expr` in the `in_assert` state. The `in_assert` state uses *metal*'s generic type "`any`" to match assignments, and pointer increments and decrements of any type. Note that the assignment operator will also detect uses of C's infix operators (e.g., `+=`, `-=`, etc.). The extension matches any function call with any set of arguments using the extended types `any_call` and `any_args` in combination. To assist developers in writing extensions, *metal* provides a set of generic types for matching different classes of types (e.g., scalars, pointers, floats), and different programming constructs (e.g., case labels, indirections).

[2] Since patterns can match nearly arbitrary C code, it does not matter if `assert` is a function or a macro; we have modified the pre-processor to ignore line and file directives.

```
{ #include <assert.h> }
// Apply SM ignoring control flow
sm Assert flow_insensitive {
  // Match expressions of "any" type
  decl { any } expr, x, y, z;
  // Used in combination to match all
  // calls with any arguments
  decl { any_call } any_fcall;
  decl { any_args } args;

  // Find all assert calls.  Then apply
  // SM to "expr" in state "in_assert."
  start: { assert(expr); } ==>
       { mgk_expr_recurse(expr, in_assert); } ;
  // Find all side-effects
  in_assert:
     // Match all calls
     { any_fcall(args) } ==>
          { err("function call"); }
      // Match any assignment (including
      // the operators +=, -=, etc.)
    | { x = y }  ==> { err("assignment"); }
      // Match all increments and decrements
      // --z and ++z ommited for brevity
    | { z++ } ==> { err("post-increment"); }
    | { z-- } ==> { err("post-decrement"); } ;
}
```

Figure 3: A *metal* SM that warns of side-effects in `assert` uses.

When applied to Xok's ExOS library operating system, this 25 line extension found 16 violations in 199 `assert` uses. Two were false positives triggered by debugging functions. These could be suppressed by wrapping such calls in a differently named, unchecked assertion macro. The remaining fourteen cases were errors in crucial system code that would function incorrectly if the assertion was removed. The underlying cause of these errors was `assert`'s use as shorthand for checking the result of possibly-failing operations such as insertion of page table entries and deallocation of shared memory regions. A typical example is the following snippet from the ExOS "`mmap`" code to insert a page table entry:

```
/* libexos/os/mmap.c:mmap_fault_handler:410 */
assert(_exos_self_insert_pte(0, PG_P|
     PG_U|PG_W, PGROUNDDOWN(va), 0, NULL) == 0);
```

The effect of removing the assert condition (and hence these calls) would be mysterious virtual memory errors.

4.2 Checking assertions statically

Assertions specify conditions that the programmer believes must hold. Without MC, compilers are oblivious to this fact, so `assert` checks can only occur dynamically. With MC, it is possible to find errors by evaluating these conditions statically, thereby quickly and precisely finding errors.

We wrote such an extension on top of *xg++*. At a high level, it uses *xg++*'s dataflow routines to track the values of scalar variables. At each `assert` use, it evaluates the assertion expression against these known set of values. If the expression could fail, it emits a warning. Currently, *xg++* only performs primitive analysis that tracks the set of constant assignments to scalar variables on a given path. The set of possible values for a variable is then just the union of constant assignments to that variable before it is used. If any non-constant assignments occur, the value is considered "unknown." Returning the set of possible values allows the effectiveness of the checker to transparently increase as our analysis in *xg++* becomes more powerful. As a practical refinement, we eliminate a large class of false positives by ignoring assertions of the constant "0" (which always fails) since this is an idiomatic method for programmers to terminate execution in "impossible" situations.

When applied to the FLASH cache coherence code (discussed more in Section 8) the 100 line extension found five errors that could have crashed the system. These errors underscore the value of static evaluation, since they were in code that had been heavily tested for over five years. They had been missed because the length and complexity of typical FLASH code paths caused them to only occur sporadically. This complexity also makes manual detection of errors difficult. On one path, the assignment and the assertion that it violated were 300 lines apart and separated by 20 if statements, 6 else clauses, and 10 conditional compilation directives. Another case beat this by having 21 if statements, 4 else clauses, and 29 conditional compilations! Even given the exact situation that leads to the error, inspecting such paths is mind-numbing.

4.3 Discussion

Library implementations cannot inspect the context in which they are used or how they are invoked. MC can be used to attack these blindnesses. Our first extension used MC to to detect illegal actions in `assert` uses, something that an `assert` implementation cannot otherwise do either dynamically or statically. Our second extension used context knowledge to push dynamically evaluated conditions to compile time. A similar approach can be used to make certain dynamic error checks static or to improve performance by allowing implementations to specialize themselves to a given context, such as a memory allocator that generates specialized inline allocations for constant size allocation requests.

The restriction on side-effects in assertion conditions is a miniature example of a more general pattern of "language subsetting," where systems impose an execution context more restrictive than the base language in which code is written. We have built two other extensions that enforce system-specific execution restrictions. The first warns when kernel code uses floating point. It found one case where a Linux graphics driver assumes that floating point calculations will be evaluated at compile time. Using a compiler other than `gcc` or lower optimization levels could violate this assumption. The second checks for stack overflow. It found 10 places where Linux code allocated variables larger than 3K on the 6K kernel stack, and numerous 1K or larger allocations. Most of these led to patches by kernel maintainers. It also found a similar case in Xok where an innocent looking stack-allocated structure turned out to be over 8K bytes.

In addition to checking, systems can use restriction checkers for optimization by detecting when an application's actions are more limited than the general case. For example, a threads package can use smaller stack sizes than the default if it can derive an upper bound on stack usage.

5 Temporal Orderings

Many system operations must (or must not) happen in sequence. Sequencing rules are well-suited for compiler checking since sequences are frequently encoded as literal procedure calls in code. This allows a *metal* extension to find violations by searching for operations and transitioning to states that allow, disallow, or require other operations. This section discusses two such extensions. The first enforces an "X before Y" rule that system calls properly check application pointers passed to them for validity before using them. The second checks that code obeys a set of ordering rules for memory allocation and deallocation.

5.1 Checking copyin/copyout

Most operating systems guard against application corruption of kernel memory by, in part, using special routines to check system call input pointers and to move data between user and kernel space. We present an MC extension that finds errors in such code by finding paths where an application pointer is used before passing through such routines. At each system call definition, the extension uses a special *metal* pattern to find every pointer parameter, which it binds to a `tainted` state. (The use of per-variable state differs from the previous checkers that used a single, global state per path.) The only legal operations on a `tainted` variable are being (1) killed by an assignment or (2) passed as an argument to functions expecting tainted inputs (e.g, data movement routines or output functions such as `kprintf`). All other uses will be signaled as an error.

We tailored a version of this checker for the Xok exokernel code. It detects which procedures are system calls using the exokernel naming convention that such routine names begin with the prefix "`sys_`." As a refinement, the checker warns when any non-system-call routines use "paranoid" user-data routines. It examined 187 distinct user pointers in the exokernel proper and device code and found 18 errors. A typical error is this command to issue disk requests:

```
/* from sys/kern/disk.c */
int sys_disk_request (u_int sn, struct Xn_name
        *xn_user, struct buf *reqbp, u_int k) {
  ...
  /* bypass for direct scsi commands */
  if (reqbp->b_flags & B_SCSICMD)
    return sys_disk_scsicmd (sn, k, reqbp);
```

Here, the pointer, `reqbp`, is passed in from user space and dereferenced in the `if` statement without being checked.

This extension also signalled 15 false positives. Four of these were due to a stylized use where non-null pointers were verified using standard routines, but null ones were allowed through (they would be handled correctly by lower levels). Three others were due to kernel backdoors used to let system calls call other system calls with unchecked parameters. The remaining were due to the checker's lack of global analysis and its disallowing of tainted variable copies.

5.2 Checking memory management

Most kernel code uses memory managers based loosely on the C procedures `malloc` and `free`. We present an extension that checks four common rules:

1. Since memory allocation can fail, kernel code must check whether the returned pointer is valid (i.e., not null) before using it.
2. Memory cannot be used after it has been freed.
3. Paths that allocate memory and then abort with an error should typically deallocate this memory before returning.
4. The size of allocated memory cannot be less than the size of the object the assigned pointer holds.

	Linux		OpenBSD	
Violation	**Bug**	**False**	**Bug**	**False**
No check	79	9	49	2
Error leak	44	49	3	1
Use after Free	7	3	0	0
Underflow	2	0	0	0
Total	132	61	52	3

Table 2: Error counts for Linux and OpenBSD. The checker was applied 4268 times in Linux and 464 times in OpenBSD.

Figure 4 shows a stripped-down extension that checks these rules. For space, the size check and most error reporting code is omitted. This extension, like the previous one, associates each variable with a state encoding what operations are legal on it. Pointers to allocated storage can be in exactly one of four states: `unknown`, `null`, `not_null`, or `freed`. A variable is bound to the `unknown` state at every allocation site. When an `unknown` variable is compared to null (e.g., in C, "0") the extension sets the variable's state on the true (null) path to `null` and on the false (non-null) path to `not_null`. When the variable is compared to non-null, these two cases are reversed. The two initial patterns recognize C's check-and-compare allocation idiom and combine these transitions with the initial variable binding. Pointers passed to `free` transition to the `freed` state. As a minor refinement, when variables are overwritten, the extension stops following them by transitioning to the special *metal* state, `stop`.

The checker only allows dereferences of `not_null` pointers. This restriction catches instances when memory is used before being checked, on null paths, or after being freed. It catches double-free errors by warning when `freed` pointers are passed to `free`. It catches cases when error paths do not free allocated memory by warning when any `non_null` or `unchecked` variable reaches a return of a negative integer, which idiomatically signals an error path.

The full version of the checker is 60 lines of code. We get a lot for so little: the extension implements a flow-sensitive compiler analysis pass that checks for rules on all paths and takes into consideration the observations furnished by passing through conditionals. As Table 2 shows, the extension found 132 errors in Linux and 51 errors in OpenBSD. It turned up 61 and 3 false positives respectively, most due to not handling variable copies, or not detecting when allocated memory would be freed by a cleanup routine.

The most common error was not checking the result of memory allocation: 79 cases in Linux, 49 in OpenBSD. In Linux, the single largest source of these errors was an allocation macro, `CODA_ALLOC`, which was widely used throughout the Coda file system code. It contains the unfortunate code:

```
/* include/linux/coda_linux.h:CODA_ALLOC */
ptr = (cast)vmalloc((unsigned long) size);
...
if (ptr == 0)
  printk("kernel malloc returns 0 at %s:%d\n",
               __FILE__,__LINE__);
memset( ptr, 0, size );
```

While this code prints a helpful message on every failed allocation, the initialization using `memset` will immediately cause a kernel segmentation fault.

The next most common error was not freeing memory on error paths (44 in Linux, 3 in OpenBSD). A typical not-freeing error is given in Figure 5. An idiomatic mistake was to have many exit points from a function, but forgetting to free the memory at all of these points.

The seven use-after-freeing errors could cause non-deterministic bugs if another thread re-allocated the freed memory. The most common case was five cut-and-paste uses of the code:

```
/* drivers/isdn/pcbit:pcbit_init_dev */
kfree(dev);
iounmap((unsigned char*)dev->sh_mem);
release_mem_region(dev->ph_mem, 4096);
```

Here, the memory pointed to by `dev` is freed and then immediately used in two subsequent function calls.

Additionally, the checker discovered two under-allocation errors. These were particularly dangerous, since they could cause memory corruption whenever a routine is used, rather than only failing under high load. One was caused by an apparent typo where the size of the memory needed for a structure of type `struct atm_mpoa_qos` (92 bytes) was computed using the size of a structure of type `struct atm_qos` (84 bytes):

```
/* net/atm/mpc.c:169:atm_mpoa_add_qos */
struct atm_mpoa_qos *entry;
...
entry = kmalloc(sizeof(struct atm_qos),
                               GFP_KERNEL);
```

The other error reversed `kmalloc`'s size and interrupt level arguments, specifying that 7 (the value of `GFP_KERNEL`) bytes of storage to be allocated instead of 16. Currently, both errors are harmless, since the kernel uses a power-of-two memory allocator with a minimum allocation unit of 32 bytes. However, they are latent time bombs if a more space efficient allocator is ever used.

```
sm null_checker {
   decl { scalar } sz;      // match any scalar
   decl { const int } retv; // match const ints
   decl { any_ptr } v1;     // match any ptr
   // 'state' specifies 'v' will have a state
   state decl { any_ptr } v;

   // Associate allocated memory with unknown
   // state until compared to null.
   start, v.all:
       // set v's state on true path to "null",
       // on false path to "not_null"
       { ((v = (any)malloc(sz)) == 0) }
          ==> true=v.null, false=v.not_null
       // vice versa
     | { ((v = (any)malloc(sz)) != 0) }
          ==> true=v.not_null, false=v.null
       // unknown state until observed.
     | { v = (any)malloc(sz) } ==> v.unknown;

   // Allow comparisions on variables in
   // states "unknown", "null", and "not_null."
   v.unknown, v.null, v.not_null:
      { (v == 0) } ==>
          true = v.null, false = v.not_null
    | { (v != 0) } ==>
          true = v.not_null, false = v.null;

   // Catch error path leaks by warning when
   // a non-null, non-freed variable gets to a
   // return of a negative integer.
   v.unknown, v.not_null: { return retv; } ==>
        { if(mgk_int_cst(retv) < 0)
            err("Error path leak!"); };

   // No dereferences of null or unknown ptrs.
   v.null, v.unknown: { *(any *)v } ==>
          { err("Using ptr illegally!"); };

   // Allow free of all non-freed variables.
   v.unknown, v.null, v.not_null:
        { free(v); } ==> v.freed;

   // Check for double free and use after free.
   v.freed:
        { free(v) } ==> { err("Dup free!"); }
      | { v } ==> { err("Use-after-free!"); };

   // Overwriting v's value kills its state
   v.all: { v = v1 } ==> v.ok;
}
```

Figure 4: *Metal* extension that checks that allocated memory is (1) checked before use, (2) not used after a free, (3) not double freed, and (4) always freed on error paths (those returning a negative integer).

```
/* from drivers/char/tea6300.c */
static int tea6300_attach(...) {
  ...
  client = kmalloc(sizeof *client,GFP_KERNEL);
  if (!client)
    return -ENOMEM;
  ...
  tea = kmalloc (sizeof *tea, GFP_KERNEL);
  if (!tea)
    return -ENOMEM;
  ...
  MOD_INC_USE_COUNT;
  ...
}
```

Figure 5: Code with two errors: (1) not freeing memory (`client`) on an error path and (2) (discussed in Section 6) calling `MOD_INC_USE_COUNT` after potentially blocking memory allocation calls.

Check	Local	Global	False Pos
Interrupts	18	42	4
Spin Lock	21	42	4
Module	22	~ 53	~ 2
Total	61	~ 137	~ 10

Table 3: Results for checking if kernel routines block (1) with interrupts disabled ("Interrupts"), (2) while holding a spin lock ("Spin Lock"), or (3) in a way that causes a module race ("Module"). We divide errors into whether they needed local or global analysis. Local errors were due to direct calls to blocking functions; global errors reached a blocking routine via a multi-level call chain. The global analysis results for Module are marked as approximate since they have not been manually confirmed.

While these checks focus on raw byte memory management, the general extension template can be retrofitted to check similar rules for other, higher-level objects. A modified version of this extension found 15 probable errors in Linux "IRQ" allocation code where allocations were not checked for errors, and IRQ's were not deallocated on error paths.

6 Enforcing Rules Globally

The extensions described thus far have been implemented as local analyses. However, many systems rules are context dependent and apply globally across functions in a given call chain. This section presents two extensions that use *xg++*'s global analysis framework to check the following Linux rules:

1. Kernel code cannot call blocking functions with interrupts disabled or while holding a spin lock. Violating this rule can lead to deadlock [28].
2. A dynamically loaded kernel module cannot call blocking functions until the module's reference count has been properly set. Violating this rule leads to a race condition where the module could be unloaded while still in use [26].

We first describe a global analysis pass that computes a transitive closure of all potentially blocking routines. Then, we discuss how the two extensions use this result.

6.1 Computing blocking routines

We build a list of possibly blocking functions in two passes. The first, local pass, is a *metal* extension that traverses over every kernel routine, marking it if it calls functions known to potentially block. In Linux, blocking functions are primarily (1) kernel memory allocators called without the `GFP_ATOMIC` flag (which specifies not to sleep when the request cannot be fulfilled) or (2) routines to move data to or from user space (these block on a page fault). After processing each routine, the extension calls *xg++* support routines to emit the routine's flow graph to a file. The flow graph contains (1) the routine's annotation (if any) and (2) all procedures the routine calls. After the entire kernel has been processed, each input source file will have a corresponding emitted flow graph file. The second, global pass, uses *xg++* routines to link together all these files into a global call graph for the entire kernel. The global pass then uses *xg++* routines to perform a depth first traversal over this call graph calculating which routines have any path to a potentially blocking function. The output of this pass is a text file containing the names of all functions that could ever call a blocking function. Running the global analysis on the Linux kernel gives roughly 3000 functions that could potentially sleep.

6.2 Checking for blocking deadlock

Linux, like many OSes, uses a combination of interrupt disabling and spin locks for mutual exclusion. Interrupt disabling imposes an implicit rule: a thread running with interrupts disabled cannot block, since if it was the last runnable thread, the system will deadlock. Similarly, because of the implementation of Linux kernel thread scheduling, threads holding spin locks cannot block. Doing so causes deadlock when a sleeping thread holds a spin lock that a thread on the same CPU is trying to acquire.

Our *metal* extension checks both rules by assuming each routine starts in a "clean" state with interrupts enabled and no locks held. As it traverses each code path, if it hits a statement that disables interrupts, it goes to a `disabled` state; an enable interrupt call returns it to the original state. Similarly, if it hits a function that acquires a spin lock, it traverses to a `locked` state; an unlock call returns it to the clean state. While in either of these states (or their composition), the extension examines all function calls and reports an error if the call is to a function in the list of potentially blocking routines.

Despite the simplicity of these rules, real code violates it in numerous places. The extension found 123 errors in Linux. Of those errors, 79 could lead to deadlock. The remaining 44 were calls to `kmalloc` with interrupts disabled. Possibly motivated by the frequency of this error, the `kmalloc` code checks if it is called with interrupts disabled, and, if so, it prints a warning and re-enables interrupts. In situations where interrupt disabling was used for synchronization, this leads to race conditions. The following code snippet is representative of a typical error (the mistake has been annotated in the source but not fixed):

```
/* drivers/sound/midibuf.c */
save_flags(flags);
cli();
...
while (c < count)
  ...
  for (i = 0; i < n; i++)
    /* BROKE BROKE-CANT DO THIS WITH CLI!! */
    copy_from_user((char *)&tmp_data,
                            &(buf)[c],1);
    QUEUE_BYTE(midi_out_buf[dev], tmp_data);
    c++;
  }
restore_flags(flags);
```

The call to `copy_from_user` can implicitly sleep, but is called after interrupts have been disabled with the call to `cli`.

The local errors seem to be caused by driver implementors not having a clear picture of either (1) the rules they have to follow and (2) that user data movement routines can block. The global errors seem to be caused by the fact that it is often hard to tell if a function can potentially block without tediously tracing through several function calls in different files, or without a considerable amount of *a priori* Linux kernel knowledge.

The checker produced eight false positives. Six were because the global calculation of blocking functions does not check if a called function would re-enable interrupts before calling a blocking function. Two others were caused by name conflicts where a file defined and called a function with the same name as a blocking function.

The approach of this section also applies to other operating systems. Another implementor used our system to write an extension for the OpenBSD system that checked if interrupt handling code called a blocking operation. He found one bug where an interrupt handler could call a page allocation routine that in turn called a blocking memory allocator [29].

6.3 Checking module reference counts

Linux allows kernel subsystems to be dynamically loaded and unloaded. Modules have a reference count tracking the number of kernel subsystems using them. Modules increment this count during loading (using `MOD_INC_USE_COUNT`) and decrement it during unloading (using `MOD_DEC_USE_COUNT`). The kernel can unload modules with a zero reference count at any time. A module must protect against being unloaded while sleeping by incrementing its reference count before calling a blocking function. Similarly, during unloading, it cannot block after decrementing its count. Finally, if the module aborts installation after incrementing its reference count, it must decrement the count to restore it to its original value.

Our extension checks for load race conditions by tracking if a potentially blocking function has been called and flagging subsequent `MOD_INC`s. Conversely, it checks for unload race conditions by tracking if a `MOD_DEC` has been performed and flagging subsequent calls to potentially blocking functions. It finds dangling references by emitting an error when a `MOD_INC` has not been reversed along a path that returns a negative integer (which idiomatically signals an error). As Table 3 shows, a local version of the extension that did not use the global list of blocking functions found 22 rule violations, whereas the global version found 53 cases (we have not yet confirmed the global errors).

7 Linux Mutual Exclusion

The complexity of dealing with concurrency leads most of the Linux kernel and its device drivers to follow a localized strategy where critical sections begin and end within the same function body. Despite this stylized use, the size of the code and implementors' imperfect understanding leads to errors. We wrote an extended version of the interrupt checker described in Section 3 to check that each kernel function conforms to the following conditions:

Condition	Applied	Bug	False Pos
Holding lock	~ 5400	29	113 (90)
Double lock	-	1	3
Double unlock	-	1	20 (18)
Intr disabled	~ 5800	44 (43)	63 (54)
Bottom half	~ 180	4	12
Bogus flags	~ 3200	4	49 (24)
Total	-	83 (82)	260 (201)

Table 4: Results of running the Linux synchronization primitives checker on kernel version 2.3.99. The **Applied** column is an estimate of the number of times the check was applied. We skipped twelve warnings that were difficult to classify. The parenthesized numbers show the changes when the two files with the most false positives are ignored.

1. All locks acquired within the function body are released before exiting.
2. No execution paths attempt to lock or unlock the same lock twice.
3. Upon exiting, interrupts are either enabled or restored to their initial state.
4. The "bottom halves" of interrupt handlers are not disabled upon exiting.
5. Interrupt flags are saved before they are restored.

Table 4 shows the results of running the extension on Linux. The "Applied" column is an estimate of the number of times each check was applied. Two device drivers account for a large number of false positives because they use macros that consult runtime state before locking or unlocking. The parenthesized numbers show the changes in the false positive results (over 20%) when these two files are ignored.

The most common bugs are either holding a lock or leaving interrupts disabled on function exit. These bugs often occur when detecting an error condition after which the function returns immediately. For example, the checker found this bug in a device driver for PCMCIA card services

```
/* drivers/pcmcia/cs.c:
            pcmcia_deregister_client */
spin_lock_irqsave(&s->lock, flags);
client = &s->clients;
while ((*client) && ((*client) != handle))
    client = &(*client)->next;
if (*client == NULL)
    /* forgot about &s->lock, flags! */
    return CS_BAD_HANDLE;
```

The checks for Linux locking conventions have resulted in seven kernel patches, including a fix for the error shown above. All seven patches fix cases where a lock is mistakenly held when exiting a function, and six of the seven are in device drivers (the last patch was to an implementation of ipv4 network filters). We have not been able to confirm many of the other potential bugs with kernel or device driver developers, though several strong OS implementors have examined them and consider them to be at least suspicious. Most of the potential bugs are in device drivers and networking code – this is not surprising since much of this code is written by developers throughout the world with varying degrees of familiarity with the Linux kernel.

The false positives mostly come from three sources. Code that intentionally violates the convention for the sake of efficiency or modularity accounts for 90 false positives. For example, sometimes a family of related device drivers will define an interface that breaks the conventions. Another large source of false positives (48) is caused by the fact that our checker only performs local analysis. Some drivers implement their own locking functions using the basic primitives provided by the system. The checker will warn when these functions exit holding a lock or with interrupts disabled, which is exactly what they are supposed to do. Global analysis could eliminate many of these false positives. Finally, the fact that our system does not prune simple, impossible paths accounts for 35 false positives. A typical example of this is when kernel code conditionally acquires a lock, performs an action, and then releases the lock based on the same condition. There are only two possible paths through this code, not the four that our system thinks exist.

The remaining 21 false positives could be eliminated by extending the checker's notion of locking functions and changing our system to prune the false branch of loop conditionals of the form "`for(;;)`."

8 Optimizing FLASH

In addition to checking, MC can be used for optimization. Below, we describe three extensions written to find system-level optimization opportunities in the FLASH machine's cache coherence code [20]. This code must be fast because it implements functionality (cache coherence) that is usually placed in hardware. Eliminating even a single instruction is considered beneficial. Several of the protocols examined here have been aggressively tuned for years due to their use in numerous performance papers as evidence for the effectiveness of software-controlled

Optimization	Number	False Pos	LOC
Buffer Free	11	9	30
Message Length	40	0	32
XOR Opcode	hundreds	∼10	400(*)

Table 5: MC-based FLASH optimizer results. **Number** counts how many optimization opportunities were found. The XOR checker is written in an old version of the system — a version written in *metal* would be several factors smaller.

cache coherence. Despite this effort, MC optimizers found hundreds of optimization opportunities, mostly due to the difficulty in manually performing equivalent searches across FLASH's deeply nested paths.

Buffer-free optimization. Each time a FLASH node receives a message, it invokes a customized message protocol handler that determines how to satisfy the request and update the protocol state. Handlers use the incoming message buffer to send outgoing data messages, and must free it before exiting. Handlers can send data messages, which need a buffer, and control messages, which do not. Many handlers send more than one message when responding to a request. To minimize the chance of losing a buffer, implementors are typically conservative and defer buffer freeing until the last handler send, irrespective of whether the last send(s) was a control message and therefore did not need a buffer. Unfortunately, while this strategy simplifies handler code, it increases buffer contention under high load.

Our extension indicates when buffer frees can occur earlier in the code. It traces all sends on each path through the function, and by looking at send arguments, detects if the send (1) needs a buffer and (2) frees its buffer. It gives a suggestion for any path that has an active buffer that ends with a "suffix" of control sends. The extension is 56 lines long, and found 11 instances in a large FLASH protocol, "`dyn_ptr`," where the buffer could be safely freed earlier. Each of these optimizations could be implemented by changing only two lines of code. The extension also produced nine false positives. Most of these were cases where the execution path was too complex to optimize without major code restructuring.

Redundant length assignments. Our second, lower-level optimization extension detects redundant assignments to a message buffer's length field. For speed, when sending multiple messages, implementors set a buffer's message length early in a handler and then try to reuse this setting across multiple messages. Long path lengths make it easy to miss redundant assignments. Our checker detects redundancies by recording the last assignment on every path and warning if there are two assignments of the same constant. It discovered 40 redundant assignments in the FLASH protocol code.

Efficient opcode setting. Message headers must specify the message's opcode (type). Opcode assignment costs two instructions. However, if the handler knows what opcode is currently in a header, it can change the opcode in one instruction by xoring the message header with the xor of the new and current opcode. Our extension detects such cases by computing when a message header, with known opcode, is assigned a new opcode. Both the old and new opcodes must be the same on all incoming paths. The extension determines the initial header value by looking in an automatically-built list of all opcodes a handler might receive. If there is only one possible opcode value, the extension records it and starts in a "known" state. Otherwise, the checker starts in an "unknown" state. It transitions from this state to the "known" state after the first opcode assignment. Each assignment encountered in the known state is annotated with the current opcode value. A second pass then checks every assignment and, if all paths reached it in the known state with the same opcode, emits a warning to the user that xor could be used to save an instruction. This checker found hundreds of such cases.

9 Conclusion

Systems are pervaded with restrictions of what actions programmers must always or never perform, how they must order events, and which actions are legal in a given context. In many cases, these restrictions link together the entire system, creating a fragile, intricate mess. Currently, systems builders obey these restrictions as well as they can. Unfortunately, system complexity makes such obedience difficult to sustain. Programmers make mistakes, and often they have only an approximate understanding of important system restrictions. Such mistakes can easily evade testing, which rarely exercises all cases.

We have shown that many system restrictions can be automatically checked and exploited using meta-level compilation (MC). MC makes it easy for implementors to extend compilers with lightweight system-specific checkers and optimizers. Currently, a system rule must be understood by all implementors. MC allows one implementor, who understands this rule, to write a check that is enforced on everyone's code. This leverage exerts tremendous practical force on the development of complex systems.

Check	Errors	False Positives	Uses	LOC
Side-effects(§ 4.1)	14	2	199	25
Static assert(§ 4.2)	5	0	1759	100
Stack check(§ 4.3)	10+	0	332K	53
User-ptr(§ 5.1)	18	15	187	68
Allocation(§ 5.2)	184	64	4732	60
Block(§ 6.2)	123	8	-	131
Module(§ 6.3)	~75	2	-	133
Mutex(§ 7)	82	201	14K	64
Total	~511	~292	-	669

Table 6: The results of MC-based checkers summarized over all checks. **Error** is the number of errors found, **False Positives** is the number of false positives, **Uses** is the number of times the check was applied, and **LOC** is the number of lines of *metal* code for the extension (including comments and whitespace).

MC is a general approach, scaling from simple cases such as checking assertions up to global strategies for mutual exclusion and deadlock avoidance. We have demonstrated MC's power by using it to check four real, heavily-used, and tested systems. It found bugs in all of them — roughly 500 in all — many of which would be difficult to find with testing or manual inspection. Further, these extensions typically required less than a day and a hundred lines of code to implement. Curiously, writing code to check restrictions is significantly easier than writing code that obeys them. With few exceptions, our extensions were written by programmers who, at best, only had a passing familiarity with the systems to which they were applied. We believe that these results show that the use of meta-level compilation can significantly aid system construction.

10 Acknowledgements

We thank David Dill for many discussions on software checking, Wilson Hsieh for discussions about the initial *metal* language, Mark Heinrich for his willingness to validate our FLASH results. Wallace Huang and Yu Ping Hu verified many of the error messages from the null checker in Section 5. Rusty Russell, Tim Waugh, Alan Cox, and David Miller answered numerous Linux questions and gave feedback on error reports. Additionally, we thank the other generous readers of `linux-kernel` for their feedback and support. We thank Mark Horowitz for his support, and the initial suggestion of looking at FLASH, which led to numerous other interesting directions. Finally, we thank David Dill, Wilson Hsieh, Mark Horowitz, the anonymous reviewers and especially Andrew Myers for valuable feedback on the paper.

References

[1] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

[2] R. S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 1(43):166–192, January 1996.

[3] S. Chiba. A metaobject protocol for C++. In *OOPSLA 1995 Conference Proceedings Object-oriented programming systems, languages, and applications*, pages 285–299, October 1995.

[4] A. Chou, B. Chelf, D.R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. To appear in ASPLOS 2000, November 2000.

[5] A. Chou and D.R. Engler. Metal: A language and system for building lightweight, system-specific software checkers, analyzers and optimizers. Available upon request: acc@cs.stanford.edu, 2000.

[6] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[7] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the First Conference on Domain Specific Languages*, pages 229–242, October 1997.

[8] D.L. Detlefs, R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. TR SRC-159, COMPAQ SRC, December 1998.

[9] D.R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain Specific Languages*, October 1997. An extended version "Interface Compilation: Steps toward Compiling Program Interfaces as Languages" was selected to appear in IEEE Transactions on Software Engineering, May/June, 1999, Volume 25, Number 3, p 387–400.

[10] D. Evans, J. Guttag, J. Horning, and Y.M. Tan. Lclint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[11] R. W. Floyd. *Assigning meanings to programs*, pages 19–32. J.T. Schwartz, Ed. American Mathematical Society, 1967.

[12] C. W. Fraser and D. R. Hanson. *A retargetable C compiler: design and implementation*. Benjamin/Cummings Publishing Co., Redwood City, CA, 1995.

[13] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, December 1992.

[14] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. In *Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer*, 1999.

[15] Intrinsa. A technical introduction to PREfix/Enterprise. Technical report, Intrinsa Corporation, 1998.

[16] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[17] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[19] A. Kolawa and A. Hicken. Insure++: A tool to support total quality software. `www.parasoft.com/insure/papers/tech.htm`.

[20] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[21] J. Lamping, G. Kiczales, L.H. Rodriguez Jr., and E. Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 workshop on reflection and meta-level architectures*, 1992.

[22] T. Lord. Application specific static code checking for C programs: Ctool. In *˜twaddle: A Digital Zine (version 1.0)*, 1997.

[23] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[24] T.C. Mowry, A.K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[25] G. Nelson. *Techniques for program verification*. Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[26] K. Owens. "Please review all modules for unload races". Sent to `linux-kernel@vger.rutgers.edu`. Gives protocol to follow to prevent module unload races., 2000.

[27] R. Rashid. Personal communication. Microsoft's internal tool used to check violations in Windows device drivers., July 2000.

[28] P. Russell (rusty@linuxcare.com). Unreliable guide to hacking the Linux kernel. Distributed with the 2.3.99 Linux RedHat Kernel, 2000.

[29] C.P. Sapuntzakis. Personal communication. Bug in OpenBSD where an interrupt context could call blocking memory allocator, April 2000.

[30] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programming. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[31] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

[32] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.