Using Meta-level Compilation to Check FLASH Protocol Code

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

Mark Heinrich
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853, U.S.A.

Abstract

Building systems such as OS kernels and embedded software is difficult. An important source of this difficulty is the numerous rules they must obey: interrupts cannot be disabled for "too long," global variables must be protected by locks, user pointers passed to OS code must be checked for safety before use, etc. A single violation can crash the system, yet typically these invariants are unchecked, existing only on paper or in the implementor's mind.

This paper is a case study in how system implementors can use a new programming methodology, meta-level compilation (MC), to easily check such invariants. It focuses on using MC to check for errors in the code used to manage cache coherence on the FLASH shared memory multiprocessor. The only real practical method known for verifying such code is testing and simulation. We show that simple, system-specific checkers can dramatically improve this situation by statically pinpointing errors in the program source. These checkers can be written by implementors themselves and, by exploiting the system-specific information this allows, can detect errors unreachable with other methods. The checkers in this paper found 34 bugs in FLASH code despite the care used in building it and the years of testing it has undergone. Many of these errors fall in the worst category of systems bugs: those that show up sporadically only after days of continuous use. The case study is interesting because it shows that the MC approach finds serious errors in well-tested, non-toy systems code. Further, the code to find such bugs is usually 10-100 lines long, written in a few hours, and exactly locates errors that, if discovered during testing, would require several days of investigation by an experienced implementor.

The paper presents 8 checkers we wrote, their application to five different protocol implementations, and a discussion of the errors that we found.

1 Introduction

Systems software – system calls, kernel code, systems libraries – must obey a rich set of rules for correctness and performance. Currently, these rules reside mainly in the implementor's mind, or occasionally, in documentation. As a result, implementors must find violations and opportunities for optimization manually. The first situation leads to uncaught errors, the second to lost opportunities.

This paper is a case study in how system implementors can use a new programming methodology, meta-level compilation (MC), to easily check such invariants. We use MC to find errors in the code used to manage cache coherence on the FLASH shared memory multiprocessor. This code must obey numerous "systems type" rules such as: (1) do not perform floating point operations; (2) incoming data buffers must be freed along all outgoing paths; (3) buffers must be allocated before sending; (4) some operations must precede others (a send precedes a wait for a reply); and (5) stack references are not allowed in some cases. These restrictions, and those encountered in other systems, have the property that they can be described in a few words, but can cause many errors, since obeying them can require maintaining simultaneous, accurate, awareness of many hundred (or thousand) line code paths. As our results show, such manual awareness is erratic.

We show that easily written, *user-supplied* compiler extensions are able to statically pinpoint such errors in the program source. Many of these errors fall in the worst category of systems bugs: those that only show up sporadically after the system has been running continuously for days. Our most important results are that:

1. Many system invariants can be checked with simple, user-supplied compiler extensions. All of our checkers but one were less than 100 lines of code; some of the most effective are less than 20.

2. Such checkers are easily written. Ours were implemented by non-FLASH developers, who did not have a deep understanding of the system.

3. Such checkers are powerful. Ours found 34 bugs in well-tested FLASH protocol code.

In FLASH, a typical protocol implementation is 10-20K lines of code and several protocols have been in active use for many years. Some of the bugs that we found were difficult to diagnose for even experienced FLASH protocol implementors.

This paper is laid out as follows. Section 2 gives an overview of the FLASH system and Section 3 an overview of MC and the compiler we use to implement it. We then discuss our four most profitable checks.

*This research was supported in part by DARPA contract MDA904-98-C-A933 and by a Terman Fellowship. Mark Heinrich is supported in part by NSF CAREER Award CCR-9984314.

Section 4 presents a checker that ensures FLASH code correctly waits for data buffers to fill before using them. Section 5 describes a checker that enforces consistency between a message's length field and the actual data sent (this check found the most bugs in FLASH code). Section 6 presents a checker that finds when handlers disobey FLASH buffer management rules. Section 7 describes how we ensure FLASH code correctly follows network send deadlock avoidance rules. Section 8 describes how we enforce execution restrictions placed on FLASH code (e.g., that it cannot use floating point operations). Section 9 discusses our remaining three checks. Section 10 describes related work; Section 11 discusses our results; and Section 12 concludes.

2 Flash Overview

The Stanford FLASH multiprocessor [20] is a scalable cache-coherent DSM machine that implements its communication protocols in software that runs on an embedded processor in its programmable node controller, MAGIC. As shown in Figure 1, the MAGIC chip is the heart of the FLASH node, and is responsible for directing data to and from the processor, I/O, memory, and network interfaces under the control of the communication protocol running on its embedded *protocol processor*. A programmable protocol processor allows great flexibility in the type of communication protocols that can be run on the machine, and permits debugging, tuning, and even developing new protocols after the machine is built.

A programmable node controller, however, places a serious burden on the programmer to write both high-performance and correct protocol code. For every combination of incoming message type and incoming hardware interface there is a different software *handler* that is run on the protocol processor to handle the message. The collection of these handlers comprise a FLASH cache coherence protocol. Unlike code running on the main microprocessor, a tiny bug in one of the handlers can deadlock or livelock the entire machine. Because of these severe performance and correctness constraints, the FLASH protocols are subjected to rigorous simulation in the FlashLite simulator before they are run on the FLASH hardware [15].

Even though the detailed FlashLite simulation environment has been operational for several years and has found many bugs in the protocols in simulation, no protocol has booted perfectly on the hardware on the first try. This leaves the protocol designer with the daunting task of debugging the protocol on the real machine with primitive debugging tools and imperfect visibility into what is going on in the memory system. The bugs that are found in the handlers are almost always in rare corner cases in the protocol—cases that either never show up in simulation because of a lack of cycles or because the simulator itself omits certain behavior. For example, many architectural simulators do not simulate the I/O system, so the portion of the protocol dealing with cache-coherent I/O is never exercised. The structured nature of protocol code makes it a perfect case study to show the feasibility of using meta-level compilation to aid the FLASH protocol designer in both optimizing the code and finding latent, subtle bugs.

2.1 FLASH Protocols

Many different scalable cache coherence protocols exist, differing in both the data structures used to keep the sharing information (the *directory entry*) and the number and type of messages they send. The performance and scalability of several well-known protocols (bitvector/coarsevector, COMA, dynamic pointer allocation, RAC, and SCI) have been studied using FLASH [14, 16, 27]. The details of each protocol are beyond the scope of this paper, but many aspects of the protocols are the same from the perspective of the protocol designer. Namely, there are three main types of protocol handlers that can be distilled down from the 65-90 handlers that comprise a cache coherence protocol: pass-thru handlers, directory-consulting handlers, and intervention handlers. The implementation of the pass-thru handlers are short (1-3 instructions) and there is little for MC techniques to improve or correct. However, the latter two handler types involve a common set of functionality across all the FLASH protocols, and we describe their basic operation below.

Directory-consulting handlers: These are handlers that access memory on behalf of the main processor and also access the directory. The most common example is the handler for a local cache miss from the main processor. These handlers must access the directory entry, update directory state, write back the entry, send data to the proper destination, and free the *data buffer* associated with the message when it is done. In addition to properly managing buffer allocation and de-allocation, these handlers must ensure that any messages they send have a length field in the message header corresponding to the opcode of the message being sent.

Intervention handlers: These are handlers that are asking for the most recent copy of the data from either the processor or I/O subsystem. These handlers must send an intervention request to either the processor or I/O system, wait for a reply from that hardware interface, and send the appropriate response message based on the status of the intervention reply. Failure to have a one-to-one correspondence between these sends and waits can potentially deadlock the hardware or lead to undetected data corruption in the system.

Table 1 gives the size of the protocols in terms of the number of lines in all source files (excluding header files), the number of paths through every function, and the average and max length of the paths in an entire protocol. The number of paths counts the set of unique exit paths from the beginning of the function to all returns. As the figure shows, protocols are tens of thousands of lines long, with typical path lengths over a hundred lines long.

3 Meta-level Compilation Overview

This section gives an overview of meta-level compilation (MC) and the compiler system we have built to implement it.

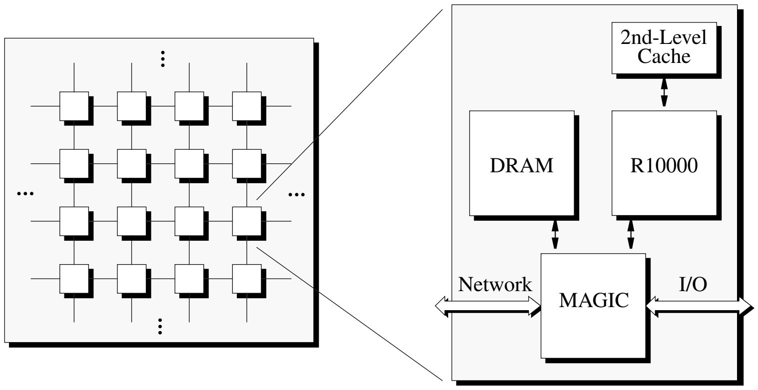

Figure 1: A FLASH node, depicting the central location of the MAGIC node controller.

Protocol	LOC	# of paths	ave/max path length
`bitvector`	10386	486	87/563
`dyn_ptr`	18438	2322	135/399
`sci`	11473	1051	73/330
`coma`	17031	1131	135/244
`rac`	14396	1364	133/516
`common_code`	8783	1165	183/461

Table 1: Protocol size as measured by lines of code (LOC), the number of unique paths from the beginning of a handler to all exit points, the average length of all paths (as LOC), and the maximum length of any path. Protocols share some common files — to avoid double-counting of these, we charged them to a special category called `common_code`.

3.1 MC motivation and methodology

The goal of meta-level compilation (MC) is to make programming significantly more powerful. It does so by raising compilation from the low-level of programming languages to the higher-level of the systems, interfaces and components they are used to implement. While MC can be used to check, transform, and optimize system-level operations, this paper focuses on checking. One intuitive way to see when MC applies in this domain is to consider the types of sentences compilers are able to enforce. Restrictions that fit these templates are good candidates for MC. Some example sentence templates, and their manifestation in this paper are:

1. "Never/Always do X": "FLASH handlers should never use floating point operations"; "handlers cannot take parameters or return results"; "the first and second statement of handler code must be calls to the special macros HANDLER_DEFS and HANDLER_PROLOGUE."

2. "Always do X before/after Y": "Before using a data buffer you must check that the hardware has finished filling it with data"; "if you allocate a buffer, you must check that allocation succeeded."

3. "If you do X, then you must/cannot do Y": "If you do a synchronous send, then you must wait for its reply"; "if you allocate a buffer you must free it"; "if you free a data buffer, you cannot use it."

Enforcing the first two types of restrictions is especially easy because they are typically value-independent requiring either no information at all, or possibly a small amount of context information, such as whether a routine is a handler or called by a handler. As we show in this paper, our system can check many instances of all three restrictions using easily expressible state machines.

3.2 *xg++* and *metal*

Our MC framework is composed of two pieces, *xg++*[9], an extensible compiler built on top of GNU `g++`, and *metal*[5], a language for writing MC extensions. *Metal* programs express program analysis passes as high-level state machines (SMs). They are dynamically linked into *xg++* and applied down every path in each function *xg++* consumes in the input source.

Metal programs define states, transitions between states, patterns that trigger transitions, and actions to be performed when a transition is triggered. The state machine part of the language is straightforward and can be viewed as syntactically similar to a "yacc" specification. Users can create arbitrary states, transition between them, and perform actions that are escapes into the base language. 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 powerful yet easy to use, since they closely mirror the source constructs they are searching for. Typically, SMs use patterns to search for interesting features, which then cause transitions between states on matches. The next two sections present checkers written in *metal* (see Figures 2 and 3).

Inter-procedural checking: 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 are then read back in and traversed. Section 7 describes a checker that uses this framework. The checkers in this paper use a simple scheme of listing functions that must obey a given restriction (e.g., free or not free a buffer). Checking that a condition holds across a procedure call boundary can then be done in two parts: the checker verifies that each caller preserves any necessary preconditions and that the procedure itself preserves the restriction.

4 Buffer Fill Race Conditions

When a node receives a message, the handler for the message begins processing the message header while the hardware interface fills the data buffer with the message body. If the handler needs to read from the data buffer, then it must explicitly call a macro that waits until the hardware has finished filling the buffer in order to avoid a race condition.

In the FLASH code, data buffers are read explicitly using the macro `MISCBUS_READ_DB`. All reads must be preceded by a call to the macro `WAIT_FOR_DB_FULL` to synchronize the buffer contents. To increase parallelism, `WAIT_FOR_DB_FULL` is only called along paths that require access to the buffer contents, and it is called as late as possible along these paths. Figure 2 shows a checker written in *metal* that examines all possible execution paths of a function for violations of this rule. (The checker used to derive the results is identical except that it also takes into account older style macros equivalent to `MISCBUS_READ_DB`.)

The checker begins by including the header file needed to define the FLASH macros and data types. The `decl` declaration specifies that `addr` and `buf` are wildcard variables that will pattern match any C integer expression ("`scalar`"). The remainder of the checker defines a simple state machine with a single state, `start`. SMs start execution in the first state they define (in this case `start`). From its start state, the SM uses two patterns to search for all uses of the macros `WAIT_FOR_DB_FULL` and `MISCBUS_READ_DB`. When either matches, the scalar expression passed as their arguments will be placed in `addr` and, for `MISCBUS_READ_DB`, `buf`. The matching rule will then cause the SM to transition to the (optional) state (the token after the `==>` operator) and then execute the (optional) action. If a rule's state is omitted, the SM remains in the current state. The `start` state has two rules. If the first rule's pattern for `WAIT_FOR_DB_FULL` matches, then the handler has correctly waited for its data buffer to fill, and any subsequent read on this execution path will be valid. Thus, the checker transitions to the `stop` state, which causes it to stop running on the current path. If the second rule's pattern matches, then the execution path being checked did not wait for its buffer to fill and it had a buffer race condition error. This rule's associated action will then print out an error message. Since the rule does not give a transition state, the checker will remain in the `start` state to catch further violations along the path.

Results: Table 2 summarizes the results of the buffer race condition checker when applied to the five FLASH protocols and common code. The checker found 4 violations in `bitvector` in rare corner cases. For example, in a couple of cases only the first byte of the buffer was read without explicit synchronization, but after an analysis of the MAGIC chip implementation we determined that they were indeed possible race conditions. The checker also produced 1 false positive in `common_code` that intentionally violated the invariant for debugging purposes.

```
{ #include "flash-includes.h" }
sm wait_for_db {
 /* Declare two variables 'addr' and 'buf' that can
  * match any integer expression. */
 decl { scalar } addr, buf;
 /* Checker begins in the first state (here 'start').
  * This state searches for two patterns conjoined
  * with the '|' operator. */
 start:
 /* The handler is allowed to read the data buffer
  * after calling 'WAIT_FOR_DB_FULL' --- once the
  * pattern below matches, we transition to the
  * 'stop' state, which stops checking on this
  * path. */
    { WAIT_FOR_DB_FULL(addr); } ==> stop
 /* If we hit a read of the data buffer in this
  * state, the handler did not do a WAIT_FOR_DB_FULL
  * first so emit an error and continue checking. */
  | { MISCBUS_READ_DB(addr, buf); } ==>
      { err("Buffer not synchronized"); }
  ;
}
```

Figure 2: A simplified *metal* checker to find violations of the rule "`WAIT_FOR_DB_FULL` must come before `MISCBUS_READ_DB`." It searches FLASH code looking for any data buffer read (using `MISCBUS_READ_DB`) not preceded by a synchronizing wait call (using `WAIT_FOR_DB_FULL`).

Protocol	Errors	False Pos	Applied
bitvector	4	0	14
dyn_ptr	0	0	16
sci	0	0	2
coma	0	0	0
rac	0	0	10
common_code	0	1	17
total	4	1	59

Table 2: The results of the buffer race condition checker. **Errors** gives the number of errors, and **False Pos** the number of false positives. The **Applied** column is the number of reads performed, to give a rough sense of the number of times the checker was applied.

5 Consistency of Decoupled Message Length State

Each time a handler sends a message, it must pass a parameter specifying if the message contains data. There is also a length field in the message header that is used by a different part of the hardware interface to determine how much message data to send. If there is no data to send, the length must be set to `LEN_NODATA`; if there

Protocol	Errors	False Pos	Applied
bitv	3	0	205
dyn_ptr	7	0	316
sci	0	0	308
coma	0	2	302
rac	8	0	346
common_code	0	0	73
total	18	2	1550

Table 3: The results of applying the message length checker to five FLASH protocols. We recorded the number of errors found (18 in total), the number of false positives (two), and and the number of times each check was applied in each protocol.

```
{ #include "flash-includes.h" }
sm msglen_check {
 /* Named patterns specifying message length assignments
  * zero and non-zero values. */
 pat zero_assign =
    { HANDLER_GLOBALS(header.nh.len) = LEN_NODATA }
  ;
 pat nonzero_assign =
    { HANDLER_GLOBALS(header.nh.len) = LEN_WORD }
  | { HANDLER_GLOBALS(header.nh.len) = LEN_CACHELINE }
  ;

 /* Named patterns specifying sends that transmit data
  * (these need a non-zero length field). */
 decl { unsigned } keep, swap, wait, dec, null, type;
 pat send_data =
    { PI_SEND(F_DATA, keep, swap, wait, dec, null) }
  | { IO_SEND(F_DATA, keep, swap, wait, dec, null) }
  | { NI_SEND(type, F_DATA, keep, wait, dec, null) }
  ;
 /* Named patterns for sends without data
  * (these need a zero length field). */
 pat send_nodata =
    { PI_SEND(F_NODATA, keep, swap, wait, dec, null) }
  | { IO_SEND(F_NODATA, keep, swap, wait, dec, null) }
  | { NI_SEND(type, F_NODATA, keep, wait, dec, null) }
  ;

 /* Start state.  Note, rules in the special 'all'
  * state are always run no matter what state the
  * SM is in.  We assume sends in this state are
  * ok and ignore them. */
 all: zero_assign    ==> zero_len
    | nonzero_assign ==> nonzero_len
    ;
 /* If we have a zero-length, cannot send data */
 zero_len: send_data ==>
    { err("data send, zero len"); }
    ;
 /* If we have a non-zero length, must send data */
 nonzero_len: send_nodata ==>
    { err("nodata send, nonzero len"); }
    ;
}
```

Figure 3: Checker written in *metal* to catch inconsistencies between message send has_data parameter and message length: data sends must have non-zero length fields, no-data sends must have zero length fields.

is data, then the length must be set to either `LEN_WORD` or `LEN_CACHELINE`.

Message lengths and the `has_data` parameter of a message send are decoupled because it simplifies the hardware design. Unfortunately, the programmer can easily assume the wrong value of the length field. This happens frequently for several reasons:

1. The protocol handlers are written in a monolithic way to improve performance at the cost of large handler functions and complex control flow. It is very tedious and error-prone to manually check the length assignments on all of the possible paths that lead to each send. It is not unusual for a length assignment to be hundreds of lines away from the message send that uses it.

2. Programmers writing FLASH protocol handlers often assume that the length need not be set if the `has_data` parameter says that there is no data for the message.

3. Each handler starts life by receiving a particular type of message. The incoming message may have the same length field value as the message being sent out in response, in which case setting the length again is redundant. If the handler assumes the wrong value for the length, it may send data with an inconsistent `has_data` parameter.

The message length field is always assigned to a constant in the FLASH code, so checking consistency with the `has_data` parameter requires knowing only the starting value of the length field and the last assignment before each send.

Figure 3 shows a checker for message length/data parameter consistency written in *metal*. Most of the checker consists of patterns that specify what constitutes a zero length assignment, a nonzero length assignment, a message send with no data, and a message send with data. The `decl` variables are wildcards; they match anything of the specified type and in this checker they simply allow the patterns to match message sends without regard to the values of those parameters.

The transitions specified at the bottom of the checker keep track of the last length assignment before each send. For simplicity, the checker shown does not consult a table or perform global analysis for the initial message length value for each handler; instead, it starts in the special state `all` that does not warn about any message sends. The only special feature of the `all` state is that its transitions are implicitly applied to other states. For example, the transitions on the patterns `zero_assign` and `nonzero_assign` also apply in the states `zero_len` and `nonzero_len`.

Table 3 shows the results of running this check on five FLASH protocols. In `dyn_ptr` and `rac`, the checker found 1 error in an "eager mode" handler, which is only used in simulation, and 6 errors in handlers for uncached reads. Uncached reads are a rare case in the protocol, so these handlers are not heavily tested. In order to trigger the bug in these handlers, the data would have to be dirty in another node's cache concurrent with a particular queue being full on the local node. This sort of bug might never occur in practice, but if it ever did it would be difficult to reproduce and diagnose. The checker also found a similar bug that exists only in `rac`.

In `bitvector` the checker found one bug in a handler for uncached reads, similar to the ones found in `dyn_ptr` and `rac`. Another bug was found in an "eager mode" handler. This bug was difficult for the author of the handler to diagnose completely, even knowing its exact location. The last bug found in `bitvector` was a clear violation of the length/data consistency rule but posed no problem during execution because of a hardware implementation detail. However, we counted the violation as a bug because it would cause problems during simulation.

Two false positives were found in the `coma` protocol in the same function. These occurred because the handler used a variable to determine the correct send parameter at run time. The variable usage was simple enough that the checker could have statically pruned the impossible execution paths with a more elaborate analysis, but the effort seemed unjustified in this case.

6 Checking Buffer Management

Every FLASH node has a set of data buffers. These are managed using manual reference counting. When a message arrives, the hardware allocates a data buffer, increments the buffer's reference count, and jumps to the appropriate message handler. This handler is responsible for decrementing the buffer's reference count when it finishes using it. Deallocation is complicated by the twin facts that (1) message handlers can be many thousands of lines long and (2) the same buffer is used to send outgoing messages and can be implicitly held onto for use by another handler. This management approach is vulnerable to the usual problems of manual reference counting. They manifest in protocol code as follows:

1. Not decrementing a buffer's reference count. This happens when a handler either completes without an explicit deallocation, or overwrites the "current buffer pointer" with a newly allocated buffer before freeing the first. Once all data buffers have been lost, a node cannot buffer incoming messages, and the system typically deadlocks or crashes.

2. Using a buffer after freeing it or using a buffer that has never been allocated. Buffer contents can change non-deterministically if another message arrives and allocates the same buffer.

3. Double freeing the same buffer. This error is the most common that we found and has the same effect as above.

Buffer management bugs are a constant source of problems in FLASH protocols. They are also difficult to track down since they tend to come up on obscure, infrequently executed paths that cause the system to have a low-grade buffer leak that only deadlocks the system after several days. Diagnosing a buffer leak is difficult since all buffer allocation bugs lead to the same set of effects (deadlocks or crashes). Determining the set of code paths out of thousands that led to the problem is non-trivial.

Despite the complexity of the code that implements manual buffer management, it is surprisingly simple to check with a compiler. A conservative approach verifies that handlers obey the following four rules:

1. Hardware handlers begin execution with a data buffer that they must free.

2. Software handlers begin execution without a data buffer and must allocate a buffer before sending messages.

3. After a buffer is freed, no send can occur until another buffer is allocated.

4. Once a buffer is allocated, it must be freed before another buffer can be allocated.

Handlers that obey these rules on all paths cannot leak buffers. We wrote a *metal* extension that is a straightforward transliteration of them. Not counting the code to determine if a routine is a handler or not, the extension is less than 100 lines. The extension is called on every routine definition. If the routine being checked is a hardware handler, the SM begins in a "has buffer" state, otherwise it begins in the "has no buffer" state. A free causes a transition from the "has buffer" state to the "has no buffer" state, whereas an allocation goes in the opposite direction. For the "has buffer" state, allocations or returns are signalled as errors, whereas the "has no buffer" state gives errors for frees and buffer uses. Frees can either be explicit or caused by calls to routines that expect buffers and free them. The extension keeps a table of routines in the latter category so that it can check them for consistency. Buffers can be used by explicit sends or by calls to routines that expect them. The SM checks for these routines using a another table, which also lets it check for consistency.

6.1 Results

Table 4 shows the results of applying the buffer management checker to the five protocols. It lists the number of errors we caught; cases which were technically errors or violations but were either minor or happened on unreachable code paths; and the number of useful and useless annotations (discussed below).

Errors: Our checker found 9 errors. We found two double free bugs in `rac`, two in `bitvector`, and two very rare ones in `dyn_ptr`. We found 3 minor errors in `sci`, two double frees and one leak. The errors are in code that is not yet part of the production protocol. Except for one case, all buffer allocation problems were double frees.

The code used by these protocols shares a common legacy. As a result, an error in the parent source gets propagated faithfully to all of the others: `dyn_ptr`, `rac` and `bitvector` all share a similar bug because of their common heritage. The interesting thing about this bug is that it was fixed in the original source, but the maintainer did not know to update the other protocols.

Violations: Violations correspond to abstraction errors, bugs in unreachable handlers, and harmless violations. `sci` has 2 of these, `dyn_ptr` has 2, and there is 1 in the common code. The `bitvector` protocol has a violation that could be a bug in legacy code, but this code was written by someone who left the project and the violation cannot be diagnosed by the current implementors. The protocol also had a very sloppy coding practice (not counted in violations) that guaranteed an invariant held in an obscure manner. When this was detected, it caused a day of searching for a non-existent error.

Annotations: Dealing with false positives. Many compiler problems are undecidable. In practice this means that an aggressive static checker will produce false positives. Sifting through spurious warnings is tedious and can be dangerous, since they can hide real errors. Thus, a checker must provide users with a way to turn off warnings. An easy way to do so is to supply a set of reserved functions that the user calls to assert a condition is true. Our checker defines two such functions: `has_buffer` and `no_free_needed`. When called in FLASH code, the first tells the checker there actually is a buffer, and the second that a message buffer does not need to be freed on this path. The checker can then use *xg++*'s interfaces to delete this call from the source. For the checkers we describe in this paper, the typical number of these annotations needed is roughly one per thousand lines of source.

In our results, we counted an annotation as useful if it suppressed a warning that the protocol implementor had difficulty evaluating. These serve as useful checkable comments in that the extension can warn when they are wrong (e.g., not needed on any path). There were 10 such annotations for `sci`, 3 for the common code and `dyn_ptr` and two for `rac`. These annotations occurred in special purpose paths in handlers that explicitly did not deallocate buffers so that a subsequent handler could use it.

Most useless annotations arose because we do not prune simple impossible paths. The most common case was protocol code that had an "if-else" branch on a condition, finished the branch, and then did another "if-else" branch on the same condition (or its negation). There are only two possible paths through this code rather than the four our system thinks exist. The rest of the cases were data-dependent actions that determined if a buffer was freed or not. We could have hard-wired in support for these but it was easier to just suppress them.

We eliminated over twenty useless annotations by adding twelve lines to the SM to make it sensitive to the value of four routines that, when called, returned a `0` or `1` depending on whether or not they freed a buffer. Without this addition, the more naive extension marked the buffer as freed (or not freed) on both paths, giving a small cascade of errors.

7 Deadlock Restrictions on Message Sends

FLASH nodes avoid message loss by only sending messages when space is available in the designated network output queue. Unfortunately a message send cannot simply block or wait for output queue space to become available because this in itself may deadlock the machine if a cycle of senders is waiting for this node to unblock to make progress. FLASH avoids this situation by ensuring that handlers only run when they have enough queue space to complete. Mechanically, FLASH divides the physical network into four virtual message slots ("lanes"). Each handler has a fixed set of lanes assigned to it when a protocol is designed. The hardware will not run a given handler until these slots are available. Before the handler can send more than its allowance on a given lane, it must explicitly check that there is enough space and suspend itself if there is not. This rule is a good example of a simple-to-state global restriction that, because it applies to the entire system, is depressingly difficult to manually enforce or check. It is also a good example of how MC can be applied to enforce a context-sensitive relation of the form "in situation X (after n sends on a lane) you must do Y (check for free buffers) before Z (sending)."

Protocol	Errors	Minor	Useful	Useless
`dyn_ptr`	2(rare)	2	3	3
`bitvector`	2	1	0	1
`sci`	3	2	10	10
`coma`	0	0	0	0
`rac`	2	0	2	4
`common`	0	1	3	7
`total`	9	6	18	25

Table 4: Buffer management checker results. **Errors** gives the number of actual errors found. The three errors in `sci` were in code that was only partially implemented. **Minor** a count of: abstraction errors, bugs in unreachable handlers, and harmless violations. **Useful** gives the number of good annotations (that document comments), **Useless** the number of false positives.

At a high level, the extension checks a protocol-writer supplied list of each handler's lane allowances against the maximum number of sends per lane the handler could do on any path. Since paths can span procedures, this extension uses *xg++*'s global analysis framework to compute these results in two passes. The first local pass walks over every handler annotating each send with the lane it uses. After processing each handler, the extension calls *xg++* support routines to emit the procedure's flow graph to a file. The flow graph contains (1) all of the routine's annotations (if any) and (2) all procedures the routine calls. The second, global pass, uses *xg++* routines to link together all of the files emitted by the local pass into a global call graph for the entire protocol. It then uses *xg++* routines to do a depth first traversal over this call graph, computing the maximum number of sends per lane the path can do. Any send that exceeds the handler's lane allowance is flagged. The check is roughly 220 lines of code. Much of this code deals with printing out precise textual "back traces" for the inter-procedural path that caused the problem — path length and branching complexity make this feature crucial for diagnosing errors.

For straight-line code without function pointers, this extension is conservative and sound: while there exist correct handlers it would reject, any handler that satisfies it cannot exceed its buffer quota. Most handlers fit this model, and are accepted. However, there are a few handlers that use loops or recursion. Cycles create problems for static analysis since, in general, it is impossible to determine how many iterations a cycle can cause. Fortunately, FLASH handlers have a useful, common special case: cycles that do not send. Such cycles represent a "fixed point" of behavior in that executing this cycle cannot increase the sends the function executes. Therefore, the extension can safely ignore them. The extension detects function-level fixed points by recording the set of lanes that are active when it encounters a function. If it sees this function again on a given path, it compares the current set of available buffers to this saved copy. If they are the same, then the cycle is a fixed point and the handler stops checking that path. If there were sends, then it warns of a possible error. This simple modification completely eliminates all recursion based false-positives. (A similar use of fixed point can be used for any extension that checks side-effects.)

Results The extension found two serious bugs: one in `dyn_ptr` and one in `bitvector`. The first case was caused when an implementor who had not written the protocol inserted code to workaround a hardware bug. This seems to be due both to the lack of clear documentation and to the fact that handlers are large, and their context opaque. The second bug appears to be a typo and seems due to probability: if something has to be done correctly many times, it will be done wrong eventually. Both bugs could cause sporadic deadlocks. There were no false positives.

8 Handler Execution Restrictions

Like many OS kernels and embedded systems, FLASH's execution environment is more restrictive than the language in which the code is written. Without compiler support, programmers can stray into illegal territory without warning. With compiler support, such wanderings are prevented. Restricting a base language to a subset is especially easy since no analysis or transformation is required: the extension need only check that certain constructs are not used by programmers.

Our extension checks the three most common FLASH restrictions. First, handlers take no parameters and return no results. We check that the return type of every function definition is `void` and that it has an empty parameter list. Additionally, several macros are deprecated, and we warn if they are used.

Second, FLASH code cannot perform floating point operations. The checker registers a function with *xg++* that is invoked on every tree node and checks that no tree node has a floating point type.

Third, handlers can assert that they do not need the stack. We check that they do not cause stack accesses by verifying that non-stack handlers do not take the address of any of their local variables, that they do not allocate "too many" local variables, and that they do not declare arrays or structures larger than 64 bits (smaller structures safely reside in registers). Additionally, we enforce the documented requirement that there is exactly one "no stack" annotation at the beginning of the handler.

Protocol	Violations	Handlers	Vars
`dyn_ptr`	4	227	768
`bitvector`	2	168	489
`sci`	0	214	794
`coma`	3	193	648
`rac`	2	200	668
`common_code`	0	62	398
`total`	11	1064	3765

Table 5: Results from applying the execution restriction checker to the five FLASH protocols. Columns indicate the number of violations caught, and the number of routines and variables checked.

While they are disallowed from using the stack, such handlers can call other handlers as long as they invoke the `SET_STACKPTR` macro immediately before the call so the callee can build a stack frame. The checker verifies that: (1) there are no spurious `SET_STACKPTR` uses (i.e., every one is followed by a call) and (2) every call is preceded by a `SET_STACKPTR` invocation.

Finally, protocol code must run both on the hardware and on a simulator. The simulator requires that programmers manually insert hooks into their source that call back into the simulator on "interesting" events. The most common hooks are macros inserted at the beginning of every function to declare simulation variables and to inform the simulator whether the current function is a normal procedure, a handler, or a software handler. Without compiler support, diagnosing the omission of hooks is difficult because only simulation results are affected and no overt error is reported.

Our checker detects such omissions in two parts. First, it automatically constructs a list of all hardware handlers and software handlers by extracting the former from the protocol specification and the latter from the protocol code. During simulation, the first statement in each of these functions must be a call to the simulator informing it that a handler is about to run. All other routines must have a similar call to tell the simulator that a normal subroutine is about to run. The checker verifies that the first and second statements in a procedure are calls to the appropriate simulation hooks using the previously constructed table of handlers.

Table 5 presents the results of applying the execution restriction checker to the five FLASH protocols. The only errors found were omissions of appropriate simulator hooks. These occurred four times in `dyn_ptr`, two times in `bitvector`, three times in `coma`, and twice in `rac`. There were three violations in `sci`, but we did not count them since they occurred in unimplemented routines which caused a fatal error if called. There were no violations in the common code. None of our error counts include the instances when hooks were omitted from inline functions, even though they were, strictly speaking, violations. We give a rough feel for how applicable the check is by tracking the number of routines and variables it checked.

9 Other Checks

Three of our checks found very few errors in FLASH code. The three checks were for: handling failure of buffer allocation, manual directory entry updates, and correctly waiting on synchronous sends. While finding bugs is good, so is giving assurance that code does not contain a given type of bug. Table 6 crudely quantifies the importance of each check by counting the number of times it was applied in each protocol.

An important difference between these checks and the preceding ones is that the actions they couple tend to be located closer to each other. For FLASH code, the chance of error appears to increase in proportion to the edit distance between points. We describe the checks and their applicability in more detail below.

Data buffer allocation: After a handler has freed its buffer, it must explicitly allocate another before it can send a message with data. A buffer allocation can fail if none are available, so every allocation must check the returned buffer for an error flag before writing to it. We implemented a check to ensure that all allocations are checked for success before being used.

Manual directory entry updates: Each FLASH node has a directory that contains the state of all cache lines it is managing. If a handler modifies the state of a line, it must update the corresponding directory entry. Unfortunately, unlike normal variables, which are transparently moved from memory and back by the compiler, handlers must explicitly load directory state into memory, modify it there, and then explicitly write it back. Forgetting to write back modified data will leave a cache line's directory entry in a stale state. The checker is organized as a small SM that checks two conditions: (1) a directory entry is loaded before it is read or written; (2) if an entry is modified, it is subsequently written back. The latter condition is frequently violated by speculative handlers that modify the entry in anticipation of the common case and, if that does not occur, intentionally lose their modifications. The checker eliminates most of these false positives by using the fact that such speculative handlers will typically send a negative acknowledgement (NAK) reply which can be seen through the use of special constants when modifying the message header.

Send-wait errors: Handlers can send messages asynchronously or explicitly indicate that they will later wait for a reply. If a handler indicates it will wait, but does not, or waits on the wrong message interface, the machine will deadlock. Our extension detects this error by checking that (1) every send with the "wait" bit set is followed by a wait for the proper interface and (2) the handler does not issue another send before it has waited for the first one.

9.1 Results

The allocation failure check raised 2 false positives because of debugging code that printed the value of the buffer before checking for errors. The send-wait check found 8 places where the code broke an abstraction barrier, and performed "waits" without calling the interface supplied macros. While these were not errors, they would make simulation problematic, since hooks could not be inserted at the simulation spot.

While directory entry management did find one bug, it also accounted for the most false positives. This is partially because there are subroutines that handlers can call which assume that the handler calling them will write back the directory entry. We count these as false positives because the current checker requires the user to manually annotate these subroutines in order to eliminate warnings. However, these annotations can be seen as a benefit because they make it possible to ensure that handlers that call these subroutines write back the entry correctly. They also make it possible to check the subroutine itself for spurious write backs. Subroutines account for 14 of the false positives: 1 in `bitvector`, 4 in `dyn_ptr`, 5 in `coma`, and 4 in `rac`.

The remaining annotations were beneficial in that they provide checkable documentation of unusual conditions that were before only discussed in comments. These include situations where a speculative path intentionally avoids writing back the directory entry. This accounts for 3 false positives: 1 in `dyn_ptr` and 2 in `rac`. Additionally, some handlers back out of a speculatively modified directory entry without sending a NAK reply. This condition occurs in special circumstances that do not follow a particular pattern, so we chose to not to detect it. The remaining annotations arise from "abstraction errors" where the directory entry address is computed explicitly instead of by calling a specific address calculation macro. The proper fix for these locations is to use the standard macros.

10 Related Work

We proposed MC in [10, 11] and provided a simple system, `magik`, based on the `lcc` ANSI C compiler [13]. While the original papers laid out an initial intellectual framework, they provided little evaluation of the idea's effectiveness on real code. We have concurrently applied MC to check other systems rules in Linux, OpenBSD and the Xok exokernel OS [9], where we found hundreds of errors. These results, and those in this paper lead us to believe MC is a generally effective technique for checking software.

Several projects have specifically targeted cache coherence protocol verification [3, 25]. This work is largely orthogonal to ours. It verifies an abstract description of the protocol, whereas our work checks the protocol code itself. Below, we compare our work to high-level compilation, other verification approaches, and extensible compilers.

Higher-level compilation. Many projects have hard-wired specific application-level information in compilers. These projects include: the ERASER dynamic race detection checker [26]; compiler-directed prefetching and management of I/O [23]; the use of static analysis to check for security errors in privileged programs [1]; ParaSoft's Insure++ [19], which can check for Unix system call errors; and the GNU compilers' `-Wall` option, which warns about dangerous functions and questionable programming practices Locally, this paper addresses a different problem domain. More globally, these

Protocol	Buffer alloc		Directory(*)		Send-wait	
	False Pos	Applied	False Pos	Applied	False Pos	Applied
`bitvector`	0	17	3	214	2	32
`dyn_ptr`	2	19	13	382	2	38
`sci`	0	5	1	88	0	11
`coma`	0	32	5	659	0	7
`rac`	0	20	9	424	2	35
`common_code`	0	4	0	1	2	2
`total`	2	97	31	1768	8	125

Table 6: Applicability of the three less effective checks: each column gives the number of false positives and the approximate number of times the check was applied. The directory entry check found 1 bug in bitvector.

past projects look at specific problems that compilation can help, whereas MC provides a framework for applying compilation to all aspects of programming. Its extensibility lets implementors check restrictions not possible with these other approaches. The success of this prior work can be viewed as giving credence to their generalization in MC.

Systems for finding software errors. The problem of finding software errors is an old one. Most approaches center around either formal verification or type checking. We discuss each below.

Formal verification uses theorem provers/checkers [2, 12, 24] or model checkers [22, 29] to check that a specification is internally consistent. When applicable, it finds errors difficult to reach by other means. However, specifications are difficult and costly to construct. While recent work has begun attacking these problems [6, 17], it is extremely rare for software to be verified. In contrast, while our checkers use weaker analysis, they are dramatically easier to build and use. Further, they appear more effective: most verification papers find a small number of errors (0-2) whereas we found 34. Finally, specifications do not necessarily mirror the code they construct and, in reality, suffer from over-simplifications and missing features. Because MC extensions work directly with program source rather than an abstraction of it, they largely avoid such mapping problems.

Two recent strong-typing systems are the extended static type checking (ESC) project [8] and Intrinsa's PREfix [18]. While both of these systems use stronger analysis than our approach, they are limited to checking much lower-level errors: buffer overrun errors, NULL pointer dereferences, and (for ESC with special support) race conditions. Further, they are not extensible, and so cannot be tailored to system-specific restrictions.

Extensible compilation. Macro systems are the most venerable user-level code transformers. An advantage of such systems (e.g., Lisp) is their tight integration with the source language. However, macro systems are restricted to fairly localized code transformations, whereas *metal* extensions can do more powerful global analysis and transformations.

There have been a number of "open compiler" systems that allow programmers to add analysis routines, usually modeled as extensions that traverse abstract syntax trees. These include Chiba's `Open C++` [4]; Crew's Prolog-based ASTLOG [7] for walking over C abstract syntax trees; and Lord's scheme-based `ctool` [21] also

Checker	LOC	Err	False Pos
Buffer management	94	9	25
Message length	29	18	2
Lanes	220	2	0
Buffer race	12	4	1
Buffer allocation	16	0	2
Directory management	51	1	31
Send-wait	40	0	8
Execution-restriction	84	0	0
No-float	7	0	0
Total	553	34	69

Table 7: The results of the SM-based FLASH checkers summarized over all five protocols and the common code. **LOC** is the number of lines of *metal* code for the extension, **Err** is the number of errors found, and **False Pos** is the number of false positives.

used for traversing C. These extensions are limited to tree walking and do not have data flow information. As a result, they seem both less powerful than *metal* 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. To the best of our knowledge, these systems provide no experimental results, making it difficult to evaluate their effectiveness.

At a lower-level, the ATOM object code modification system [28] gives users the ability to modify object code in a clean, simple manner. By focusing on machine code, ATOM can be used in situations $xg{+}{+}$ cannot be, since we require source. However, it appears that $xg{+}{+}$'s framework makes static analysis of systems rules significantly easier than they would be in ATOM.

11 Experience

The good: As Table 7 suggests, the checkers described in this paper are short and relatively easy to build, yet they catch errors in code that has been thoroughly tested over a period of years. Typically it takes longer to examine the output and determine the cause of bugs than it does to write the check. In addition, the FLASH protocol handlers were not written with this kind of verification in mind, but the technique still applies well.

A subtle robustness of this technique is that these checkers were built by implementors largely ignorant of how FLASH works. This was possible because each invariant could be described in a few sentences and, given our framework, the implementation of these sentences was usually not difficult. In a sense, it is not surprising that writing the code to check FLASH invariants is much simpler than constructing FLASH code. It is the same difference that makes proof checkers easier to construct than theorem provers.

We initially used a technique based on searching the flow graph generated by *xg++* for our checks. A common situation was ensuring that a condition held along all paths, such as checking that buffers were deallocated on all paths. After building several checks this way we realized that many were describable as simple finite state machines. When we converted the checks to SMs, the code became easier to understand and, in several cases, shrank by a factor of two.

The subsequent development of *metal* further simplified code and typically shrank it by another factor of two (or more). *Metal*'s main contribution was allowing us to express code events in terms of patterns, which saved us most of the effort needed to manually write routines to traverse abstract syntax trees. Patterns also made it easy to recognize special cases, thereby taking into account infrequent but valid constructs that would otherwise turn into false positives.

The bad: While *xg++* was good at finding bugs in FLASH, FLASH was also good at finding bugs in *xg++*. The size and complexity of the FLASH protocols found bugs beyond the capabilities of our simple test cases. Fortunately, after an initial flurry of bug fixes, *xg++* has been quite stable.

Some modifications to the FLASH code base were necessary to check the protocol code. We changed some FLASH macros that used inline assembly for the MIPS architecture because *xg++* is hosted on x86, and `g++` has bugs associated with handling foreign assembly language. Constant folding was another problem because it made it difficult to detect uses of constants within constant expressions; they are folded by `g++` into a single constant before *xg++* has access to the AST. We finessed this issue by redefining the relevant macro constants as variables. Virtually all of the modifications we made were to macros in header files, not the protocol code proper.

As a third problem, bitter experience showed that a tool that is *almost* always trustworthy can make its betrayal that much more dangerous. If your tool detects a certain class of errors, you develop a certain class of blindness. In our case, this happened when protocol code appeared to be doing a double free of a buffer. An experienced implementor did the obvious fix of removing the deallocation but, after that, the machine would not boot. After a day of investigation it turned out that a few lines above the diagnosed error, the buffer's reference count had been manually double-incremented (for no apparent reason) using a function that was "never" used. (This was the one call in all 80K lines of FLASH protocol code.) Since our check never looked for such calls, it was of course blind to their effects. By trusting the tool, implementors became blind to them as well. After this incident, we added a check in the extension that aggressively objects to occurrences of this call.

12 Conclusion

This paper shows meta-level compilation can be an effective method for automatically finding violations of many "systems type" rules. Using simple checkers we discovered 34 bugs in the FLASH protocol code; in some cases this code had been tested for years. Furthermore, the checkers presented in this paper were written by non-FLASH developers who did not possess deep knowledge of the system, yet they found subtle bugs that were difficult for even experienced implementors to reason about. The restrictions on FLASH protocol code are typical of embedded systems and OS kernels. Our initial experiences lead us to believe that MC can be applied to this class of code and to software in general.

13 Acknowledgements

We would like to thank Joel Baxter, Jeff Gibson, David Lie, and David Ofelt for tirelessly answering our FLASH questions, and David Dill for many checking discussions. Seth Hallem and Kinshuk Govil gave invaluable, last minute, detailed proof-reading. We would especially like to thank Mark Horowitz for pointing out that FLASH would be a good place to use MC; this suggestion led to many interesting subsequent applications.

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. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a language for writing coherence protocols. In *Proceedings of the First Conference on Domain Specific Languages*, pages 51–65, October 1997.

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

[5] A. Chou and D. Engler. Metal: a language and system for building lightweight, system-specific software checkers, analyzers and optimizers. 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. Engler, B. Chelf, A.C. Chou, and S. Hallem. A simple, effective method for checking and optimizing systems using system-specific, programmer-written compiler extensions. To Appear in Operating Systems Design and Implementation (OSDI) 2000, April 2000.

[10] D.R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the First Conference on Domain Specific Languages*, October 1997.

[11] D.R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May/June 1999.

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

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

[14] M. Heinrich. *The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols*. PhD thesis, Stanford University, October 1998.

[15] M. Heinrich, D. Ofelt, M. Horowitz, and J. Hennessy. Hardware/software codesign of the stanford flash multiprocessor. *Proceedings of the IEEE Special Issue on Hardware/Software Co-design*, 85(3), March 1997.

[16] M. Heinrich, R. Soundararajan, J. Hennessy, and A. Gupta. A quantitatitve analysis of the performance and scalability of distributed shared memory cache coherence protocols. *IEEE Transactions on Computers*, 48(2):205–217, February 1999. Special Issue on Cache Memory and Related Problems.

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

[18] Intrinsa. A technical introduction to prefix/enterprise. Technical report, Intrinsa Corporation, 1998.

[19] A. Kolawa and A. Hicken. Insure++: A tool to support total quality software. `http://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] T. Lord. Application specific static code checking for C programs: Ctool. In *~twaddle: A Digital Zine (version 1.0)*, 1997.

[22] 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.

[23] 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.

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

[25] S. Park and D.L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the 8th ACM Symposium on Parallel Algorithsm and Architectures*, pages 288–296, June 1996.

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

[27] R. Soundararajan, M. Heinrich, B. Verghese, and et al. Flexible use of memory for replication/migration in cache-coherent dsm multiprocessors. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[28] 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.

[29] 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.