Efficient Program Monitoring Techniques

Harish G. Patil

Technical Report #1320

July 1996

EFFICIENT PROGRAM MONITORING TECHNIQUES

by

Harish G. Patil

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1996

Abstract

Programs need to be monitored for many reasons, including performance evaluation, correctness checking, and security. However, the cost of monitoring programs can be very high. This thesis contributes two techniques for reducing the high execution time overhead of program monitoring: 1) customization and 2) shadow processing. These techniques have been tested using a memory access monitoring system for C programs.

"Customization" reduces the cost of monitoring programs by decoupling monitoring from original computation. A user program can be customized for any desired monitoring activity by deleting computation not relevant for monitoring. The customized program is smaller, easier to analyze, and almost always faster than the original program. It can be readily instrumented to perform the desired monitoring. We have explored the use of program slicing technology for customizing C programs. Customization can cut the overhead of memory access monitoring by up to half.

"Shadow processing" hides the cost of on-line monitoring by using idle processors in multiprocessor workstations. A user program is partitioned into two run-time processes. One is the main process executing as usual, without any monitoring code. The other is a shadow process following the main process and performing the desired monitoring. One key issue in the use of shadow process is the degree to which the main process is burdened by the need to synchronize and communicate with the shadow process. We believe the

overhead to the main process must be very modest to allow routine use of shadow processing for heavily-used production programs. We therefore limit the interaction between the two processes to communicating certain irreproducible values. In our experimental shadow processing system for memory access checking the overhead to the main process is very low — almost always less than 10%. Further, since the shadow process avoids repeating some of the computations from the main program, it runs much faster than a single process performing both the computation and monitoring.

Acknowledgments

I am grateful to my advisor, Prof. Charles Fischer, for his tutelage and support for past four years. Through his constant encouragement, he helped me believe in myself.

This dissertation could not have been in the shape it is today without the constructive suggestions made by my prelim committee members: Prof. Jim Larus and Prof. Bart Miller.

Prof. Thomas Reps and Prof. Susan Horwitz readily allowed me to use the infrastructure from the Wisconsin Slicing Project. Genevieve Rosay helped me throughout the development and debugging of my slicing prototype.

I have benefited greatly from many discussions with my colleagues in the department. Krishna Kunchithapadam was always ready to answer my questions on operating systems. Steve Kurlander and T. N. Vijaykumar helped me clarify many aspects of my Ph. D. project. Todd Austin's ideas on SafeC improved my work on guarding. I thank Brian Johnson, Satish Chandra, Marc Shapiro, Guhan Viswanathan, Brad Richards, Manuvir Das and Madhusudhan Talluri for their suggestions and comments on my work.

I thank Prof. Susan Horwitz and Prof. Jim Larus for many suggestions that improved this dissertation. Questions and comments by my final committee helped me iron out some last-minute glitches.

Numerous individuals have helped me reach this point in life; I am glad I get a chance to thank some of them.

My mother, Shanta, always encouraged me to do my best; I would not be here without

the sacrifices made by her.

Namrata, my friend for seventeen years and my wife, was always there for me. She helped me survive the ups and downs of graduate studies.

My daughter, Nimisha, brought so much joy in my life in past nineteen months. She provided the crucial incentive for finishing graduate studies and for moving on.

I thank my high school teacher Mr. S. G. Koparkar for nurturing my liking for science and mathematics. I also thank my M. Tech. advisor at IIT-Bombay, Prof. D. M. Dhamdhere, for kindling my interest in research.

I dedicate this dissertation to the fond memories of my father, Mr. G. A. Patil, and my sister, Nilima.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Programs are monitored for various reasons including fault-tolerance, performance evaluation, correctness checking, and security [30]. There are three basic approaches to monitoring: hardware, software, and hybrid approaches. Hardware monitoring requires specialized hardware on which application programs run. Software monitoring requires instrumenting the application program's source code, system libraries, or compiler. Hybrid monitoring combines the hardware and software approaches.

Software monitoring approaches are generally more portable than hardware approaches and hence are more widely used. Profiling on Unix by giving compilers a special flag (typically *-p* for the analyzer *prof*) and memory access checking using Purify[11] are two widely used examples of the software monitoring approach. Other examples of software

monitoring include run-time correctness checking provided by diagnostic compilers [7]. Run-time checks can detect errors that cannot be detected at compile-time, including array bound violations, invalid pointer accesses, and use of un-initialized variables.

Unfortunately, instrumentation required for software monitoring imposes a very heavy overhead on the speed of execution of the user program. In [32], run-time checks were added to a C compiler. The code generated ran 10 times slower than the original code. Similar slowdowns are reported for commercially available run-time error checking systems such as Purify [11]. This penalty limits the usefulness of software monitoring for most kinds of performance analysis or correctness checking; especially for heavily-used production programs.The goal of this work is to speed up software monitoring so that it becomes affordable even for heavily used production programs.

1.2 Contributions

The major contributions of this thesis are two techniques to reduce the high cost of software monitoring: 1) program customization and 2) shadow processing. These techniques were mainly tested in the context of monitoring memory accesses in C programs, though they can be easily adapted to other kinds of monitoring activities.

As an experimental framework a source-to-source translation technique called "guarding" was developed for monitoring array and pointer accesses in C programs. Although the basic ideas in guarding have been previously studied, we contribute many innovations and practical solutions for handling "real" C programs. We have used guarding to uncover

many previously unreported errors in popular Unix utilities and benchmarks.

A prototype slicing tool for C programs was also developed during this work. The tool is based on a slicing back end that is an offshoot of the Wisconsin Program-Integration System [26]. We believe our use of slicing for program customization is a novel application of program slicing technology.

1.3 Customization

In practice when a user program is monitored for a particular activity the results from the original computations of the program are seldom of any immediate interest. *Customization* reduces the overhead of monitoring by decoupling monitoring from the original computation. A user program is customized for a monitoring activity by throwing away computations not relevant for the monitoring. The customized program is easier to analyze and almost always faster than the original program. We have explored using static program slicing for customization as a novel application of slicing technology. We use the activity being monitored as the slicing criteria to get an executable, sliced user program customized for monitoring. Customization can cut the overhead of memory access monitoring by up to half.

1.4 Shadow Processing

General purpose multiprocessors are becoming increasingly common. "Shadow processing" is a technique that uses pairs of processors, one running an ordinary application pro-

gram and the other monitoring the application's execution. We call the processor doing the monitoring a "shadow processor," as it "shadows" the main processor's execution. We have developed a prototype shadow processing system for memory access monitoring of C programs. Our system instruments a user program to obtain a "main process" and a "shadow process." The main process performs computations from the original program, occasionally communicating a few key values to the shadow process. The shadow process follows the main process, checking pointer and array accesses. The overhead to the main process is very low — almost always less than 10%. Further, since the shadow process avoids repeating some of the computations from the input program, it runs much faster than a single process performing both the computation and monitoring. Sometimes the shadow process can even run ahead of the main process catching errors before they actually occur.

1.5 Thesis Organization

The two monitoring techniques proposed in this work were tested in the context of memory access checking of C programs. Our experimental framework, including the details of our basic instrumentation technique for checking pointer and array accesses in C programs, is described in Chapter 2. Chapter 3 presents ways to reduce the overhead of monitoring by customizing the user program. Shadow processing, a technique for concurrent monitoring using idle processors in multiprocessor workstations, is described in Chapter 4. Conclusions and directions for future work are presented in Chapter 5. A brief

description of some of the errors discovered by our prototype guarding system is presented in the Appendix.

Chapter 2

Experimental Framework

2.1 Introduction

This chapter describes a prototype memory access monitoring system used to evaluate the effectiveness of various monitoring techniques proposed during this work. The prototype implements a source-to-source translation technique for run-time checking of array bound violations and invalid pointer accesses in C programs. We call our checking technique *guarding*. It involves creating objects called guards to verify time and space bounds for pointers and arrays in the user program. These guards are used to check legality of pointer dereferences and array accesses. An illegal access may either violate a space bound (accessing an array element past the maximum index) or a time bound (accessing memory that has been de-allocated). Austin *et al* [2] have reported a similar technique. We call a user program instrumented for guarding a *guarded program*.

2.2 Implementation

Guarding was implemented on a Sun SPARCstation running SunOS Release 5.4 (Solaris 2.4). An overview of our prototype is shown in Figure 1.

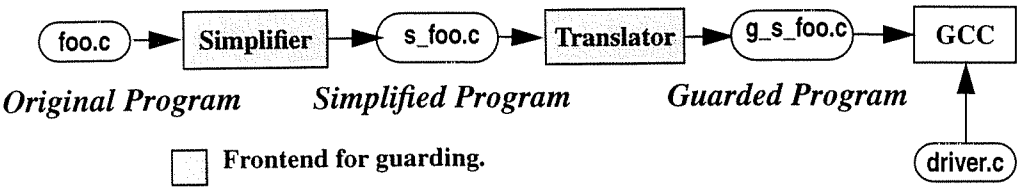


Figure 1: Overview of guarding system

Analyzing and tracking expressions involving pointers in C can be a formidable task. These expressions may involve side-effects and multiple dereferences. Further, they can occur as loop conditions, array indices, actual parameters etc. A simplification phase was introduced to restrict the case analysis required for guarding. Our simplifier is a C-to-C translator whose output is a subset of C similar to the intermediate representation called SIMPLE from McGill university[13]. Simplification greatly reduces the number of cases to be analyzed by the translator phase – there are only 15 types of basic statements in any simplified program. However a large number of temporary variables may be introduced. These variables can increase the demands on register allocation. Hence a simplified program (compiled with `-O4` using `gcc 2.5.8`) typically runs around 1-2% slower than the original input program on a Sun SPARC 630 MP.

The translator in Figure 1 reads in a simplified program and produces a guarded program. The guarded program along with a driver routine can then be compiled by a native C compiler such as *gcc*.

2.2.1 Guards

Each pointer[1] `p` in the original program has a guard `G_p` in the guarded program. The guard for a pointer stores spatial and temporal attributes for the pointer's referent. The idea is similar to safe pointers used in [2]. However we store attributes separately from the actual pointer; this allows us to concurrently monitor pointers in one process using guards in another process. Operations on pointers in the original program lead to corresponding operations on guards in the guarded program. An array of pointers in the original program has an array of guards in the guarded program. Structures and unions containing pointers have objects containing guards in the guarded program.

Valid pointers in C contain addresses of data objects (including pointers) or functions. In programs that do not cast non-pointers into pointers, the origin of a valid object pointer can be traced back to either the address-of operator, `&`, or a call to a memory allocation routine such as `malloc()`. In either case, there is an object the pointer is meant to reference. We call the object the *intended referent* of the pointer. The intended referent has a fixed size and a definite lifetime. It is clearly illegal to dereference an uninitialized pointer. Dereferencing an initialized pointer can be illegal for two reasons:

1. We will use the term pointer to include array references as well as ordinary pointers because when an array identifier appears in an expression, the array is converted from "array of T" to "pointer to T"[10].

1. The memory location being referenced is outside the intended referent of the pointer. Dereferencing the pointer will lead to a *spatial* error.

   The attributes necessary to verify that a pointer dereference is spatially valid include the number of elements in the intended referent and the current position of the pointer inside the intended referent. The actual value of the pointer is unimportant.

2. The lifetime of the intended referent has expired (e.g. the pointer points to a heap or a local object that has been freed). Dereferencing the pointer will lead to a *temporal* error.

   We allocate a unique capability (called a `key`) for each memory allocation in the original program. These keys are stored in auxiliary data structures in the guarded program. Key values change as objects are allocated and freed. Two attributes are necessary to catch temporal access errors for a pointer `p` — a copy of the key of `p`'s intended referent and a pointer to the location where the key is stored.

Given the declaration `T *ptr;` the guard `G_ptr` in the guarded program has fields storing spatial and temporal attributes for `ptr`. Fields of `G_ptr` are updated in the guarded program as the value of `ptr` changes in the original program. The fields of `G_ptr` are as follows:

- **`count`**: The number of objects of type `T` being pointed to by `ptr`. If the intended referent of `ptr` is an array, this field will hold the number of elements of the array. If `ptr` gets cast into a pointer to another object type, this field will have to be recalculated.
- **`index`**: A pointer in general may point to a collection of objects of a given type; pointer arithmetic is used to access a particular object in that collection. The field `index` in `G_ptr` denotes the offset of the current object being pointed to by `ptr`. For legal pointers, this is a non-negative value less than `G_ptr.count`. Pointer arithmetic on `ptr` leads to changes in `G_ptr.index`. *e.g.* in Figure 2, `G_p.index` is modified in the guarded program due to the statement `"p +=5"` in the original program.
- **`lock`**: An identifying code used to check the temporal legality of a dereference of `ptr`.
- **`key_Ptr`**: This field points to the identifying key of an object.This code must match

Original code	Guarding code added

int arr[10];
int *p;
 	
p = arr;
p += 5;
	a_int_guard G_arr;
	a_int_guard G_p;
	G_arr.count = 10; /* + clear other fields of G_arr*/
	/*code to clear all the fields of G_p*/
	G_p = G_arr;
	G_p.index += 5;

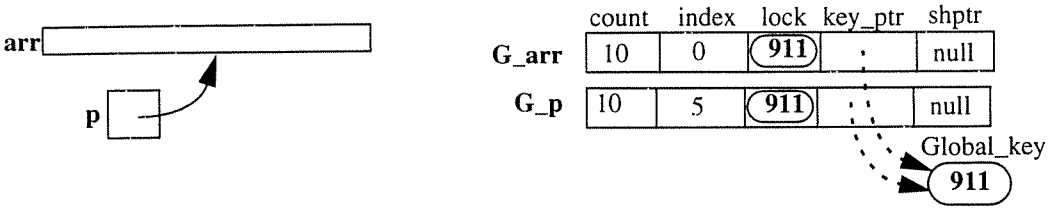


Figure 2: An example of guards for pointers and arrays

with the `lock` field of `G_ptr` for a dereference of `ptr` to be temporally valid. In Figure 2, `arr` is a global array, the field `G_arr.Key_Ptr` points to the location of the key for all global objects, `Global_Key`. Further, `G_arr.lock` has the same value (*911*) as that of `Global_Key`. After the assignment, `p = arr`, the intended referent of `p` is the same as that of `arr`, hence all the fields of `G_arr` are copied in `G_p`.

Assignment of the fields `lock` and `Key_Ptr` is discussed in Subsections 2.2.2 and 2.2.3.

- **`shptr`**: Pointers are data objects themselves, hence a pointer can reference another pointer. Each level of a multi-level pointer has a guard associated with it, and there must be a way to access each of those guards.The field `shptr` is used for that purpose. In Figure 3, the intended referent of `p` is another pointer `q`, `G_p.shptr` points to the guard of `q` *viz.* `G_q`. Thus the 2-level dereference `**p` leads to checking of two guards `G_p` and `*(G_p.shptr)` (which is `G_q`).

Original code	Guarding code added
char **p,*q, c;	**b_char_guard** G_p; /* Level 2 guard */ **a_char_guard** G_q; /* Level 1 guard */
q = &c;	/*set fields of G_q*/G_q.shptr = NULL;
p = &q;	/*set fields of G_p*/G_p.shptr = &G_q;

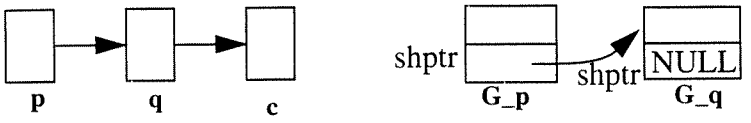


Figure 3: An example of guards for multi-level pointers.

In C, *invalid pointers* [10] can be created by casting arbitrary integer values to pointer types, by de-allocating the storage for the referent of the pointer, or by using pointer arithmetic to produce a pointer pointing outside its intended referent. It is legal to create or copy invalid pointers — attempts to dereference them are illegal. Thus pointer arithmetic and copying in the main program go unchecked.

Each pointer dereference implicit in `p[i]` or `*(p+i)` in the original program leads to two run-time checks in the guarded program:

check((unsigned)(G_p.index + i) < G_p.count)
check(G_p.lock == *(G_p.key_ptr))

The two checks are for the spatial and temporal legality of the dereference `p[i]` or `*(p+i)`. The first check is equivalent to "`0 <= (G_p.index+i) < G_p.count`."

2.2.2 Shadow heap

An auxiliary data structure called the *shadow heap* is maintained in the guarded program to help check accesses to heap objects in the original program. The shadow heap is an expandable array of unsigned integers. Each heap object in the original program has a slot in the shadow heap containing an identifying integer that is a *key* for the object. After a dynamic allocation of an object *O* in the original program, a slot from the shadow heap is reserved for *O* until the time *O* is de-allocated. This slot stores the (essentially) unique key value for *O*. A list of empty slots arising due to de-allocations of objects is maintained along with the shadow heap. These empty slots are reused later to avoid unbounded expansion of the shadow heap (much like a free-space list). When a pointer `p` points to a valid heap object *O*, `G_p.key_ptr` points to the shadow heap slot corresponding to *O*. Further, the key value in that slot matches `G_p.lock`. Key values are assigned using a global counter called *HeapKey* which wraps around to zero after reaching the maximum value (typically 2^{32}-1). Thus there is an extremely small chance (typically << 2^{-32}) that a dereference `(*p)` of a pointer whose referent has been freed will go undetected. We shall

consider the chance so small that it may be safely ignored. Actions on a dynamic allocation are explained below:

Original code	Guarding code added
p = **malloc**(S)	G_p.count = S/**aligned_sizeof**(*p) G_p.index = 0 G_p.key_ptr = **Next_heap_slot**() *(G_p.key_ptr)= G_p.lock = HeapKey++

The function **`Next_heap_slot`**`()` returns the address of an empty slot in the shadow heap. If `p` is a multi-level pointer, **`malloc`**`(S)` results in allocation of a certain number (N) of pointers. Guards for these newly allocated pointers also need to be allocated using the statement `"G_p.shptr = calloc(N,sizeof(G_p))"` where `N` is `G_p.count`. If `p` is a single level pointer, a `NULL` value is assigned to `G_p.shptr`. Calls to **`calloc`** are handled very much like calls to **`malloc`**. A call **`realloc`**`(p,newsize)` leads to changes in `G_p.count`. A call **`free`**`(p)` leads to the checking of temporal and spatial legality of `*p`. `"G_p.index > 0"` indicates `p` currently points in the middle of an object; a warning message may be printed here. In addition, freeing of non-heap objects is detected by requiring that `G_p.key_ptr` points into the shadow heap.

2.2.3 Shadow stack

In C, it is illegal to dereference a pointer to a local variable of a function that has exited. To catch these dereferences, an auxiliary data structure called a *shadow stack* is maintained. It is a stack of unsigned integers. Each active frame in the run-time stack has a slot

in the shadow stack containing its identifying key value. All local variables in a function share a slot and a key value. The key values are assigned using a global counter called *StackKey*. On a function entry, a slot gets pushed on the shadow stack, *StackKey* is incremented and its value gets stored in the newly pushed slot. On a function exit, the top slot from the shadow stack is erased and popped. After the assignment `p = &var` in the original program, `G_p.key_ptr` in the guarded program is made to point to the shadow stack slot corresponding to `var`'s enclosing frame (global variables use a special `Global_Key` which is the same as the shadow stack slot for the function `main()`). As long as the frame containing `var` is active, `G_p.lock` will continue to match the key value in the slot pointed by `G_p.key_ptr`. After the frame containing `var` is exited, its shadow stack slot will be erased. If an attempt is made to dereference `p` now, the temporal check `(G_p.lock==*(G_p.Key_Ptr))` will fail. The shadow stack slot corresponding to an exited function will get reused on the next function entry (possibly to the same function) with a different key value (*StackKey* value at that time). Hence, dereferencing `p` will continue to lead to a temporal error.

Actions after an assignment `p = &var` are explained below:

Pointer operation	Guard operation
p = &var	G_p.count = 1 G_p.index = 0 G_p.key_ptr = **<frame_slot** for var> G_p.lock = *(G_p.key_ptr)

The value <**frame_slot** `for var`> is the address of the slot corresponding to `var`'s activation record in the shadow stack. If `p` is a multi-level pointer, `var` must be a pointer with its own guard `G_var`. In this case the statement "`G_p.shptr = &G_var`" is needed. Otherwise a `NULL` value is assigned to `G_p.shptr`.

setjmp and **longjmp** functions in C implement a primitive form of non-local jumps [10]. **setjmp**`(env)` records its caller's environment in the "jump buffer" `env`, an implementation-defined array. The function **longjmp** takes as its argument a jump buffer previously filled by calling **setjmp** and restores the environment stored in that buffer. Many active frames on the stack may become inactive after a **longjmp**. We handle this by popping and erasing corresponding slots in the shadow stack before doing the corresponding **longjmp** in the guarded program.

2.2.4 Handling peculiar C language features

Our prototype requires that the input program does not cast non-pointer values into pointers. Casting a low level pointer to a higher level (*e.g.* casting an `int *` into `int **`) is also prohibited; as an exception casting `char *` returned by `malloc()` or `calloc()` to higher level pointers is allowed. We had to modify some of the test programs to handle casting of low level pointers to a higher level. Our translator currently detects pointer misuses and quits with an appropriate message. An alternative would be to flag the guard for a misused pointer to be unsafe at run-time and skip further checking for that pointer.

Multidimensional arrays in C can be reshaped (*e.g.* a two dimensional array of size `m×n` can be treated as a one dimensional array of size `m*n` or another two dimensional array of size `(m/2)×(2n)`). To uniformly handle such reshaping our simplifier converts multi dimensional arrays to one dimensional arrays. All the accesses to a multidimensional array are simplified into accesses to the corresponding uni-dimensional array.

There may be unions overlaying pointers with non-pointers in a C program. For example:

```
union {
  char *field1;
  int field2;
  struct s * field3;
}u1;
```

Suppose `u1` is a union described above. It is a "misuse" [10] to treat the value of `field2` of `u1` as `field1`. We maintain a run-time tag indicating the type of the currently active field of a union in the guarding object for the union. The guard `G_u1` for `u1` in the example above looks as follows:

```
struct {
 char * tagname; /* currently active field */
 union {
  a_char G_field1;
  a_struct_s G_field3;
  }
}G_u1;
```

`G_u1.tagname` is set appropriately whenever a field of `u1` is assigned to in the original program. A reference to a field of `u1` in the original program leads to a verification of `G_u1.tagname` in the guarded program.

Functions with pointer arguments take corresponding guards as extra arguments in the guarded program. Functions returning pointers need to also return corresponding guards in the guarded program. A guard for a returned pointer is returned indirectly using an extra parameter.

Original code	Guarded code
foo(**char** *p,**int** c);	guarded_foo(**char** *p,**int** c,**a_char_guard** G_p);
char * bar(); q = bar();	**char** * guarded_bar(**a_char_guard** *G_retval); q = guarded_bar(&G_q);

Currently, we perform checks only for user defined functions for which source code is available and we provide a clean interface for external functions. There is no passing of guards for pointer parameters to external functions. Special handling is needed for external functions returning pointers. The external functions may either be "system calls" (as described in section 2 of the UNIX man pages) or "C-library routines" (as described in section 3 of the UNIX man pages). Calls to operating system kernel routines (system calls) are one source of non-determinism for programs. A system call is a well defined entry point into operating system code. The return value of a system call depends on the state of the operating system data structures at the time the call was made. If we want the guarded program to get the exact same return values from the system calls as the original program then we will have to instrument the original program to record the return values

of system calls. From our experience with SPEC92 benchmarks, the number of values to be recorded are too few to cause any significant difference in the execution of the original program. However, we believe that letting the guarded program repeat the system calls is safe —as it will monitor *some feasible* run of the original program. Also it allows us to run a monitored program independent of the original program.

Currently, we support only single process programs. Programs calling `fork` or `exec` are not supported. We can envision creating a new guarding process for each child of the original process.

2.3 Errors uncovered

Run-time errors that do not crash programs can go unnoticed for a long time. Guarded programs report such errors as they occur. Programmers can use the feedback from the guarded programs to eliminate subtle bugs. We uncovered unreported errors in seven test programs that did not crash. These programs (with the number of errors in parentheses) were *decompress*(1) and *sc*(2) from SPEC92, *cholesky*(2) and *locus*(2) from the SPLASH benchmarks, and *cb*(1), *ptx*(4), and *ul*(1) from SunOS. Four SunOS utilities *col*, *deroff*, *uniq*, and *units* crashed with random inputs. These were already reported to be buggy [24] in earlier versions of SunOS. However, we found new errors in these utilities as well. In all, we uncovered 19 errors in eleven programs. We include a brief description of some of those errors in the Appendix. We expect further testing will reveal errors in other widely-used programs.

2.4 Performance of Basic Guarding

We use programs from the SPEC92[4] benchmarks to report performance of guarding.

Table 1: Guarding: Characteristics of test programs

Program	# of files	# of lines	Description
alvinn	1	272	Trains a neural network using back propagation to keep a vehicle from driving off a road.
compress	1	1503	A data compression application that uses Lempel-Ziv coding to compress a 1MB file. (Modified for testing.)
ear	13	5237	Uses FFTs and other library routines to simulate the human ear.
eqntott	23	3454	Translates boolean equations into truth tables.
espresso	44	14838	An EDA tool that generates and optimizes PLA structures.
sc	8	8485	A spreadsheet benchmark.
xlisp	22	7741	A LISP interpreter. (Modified for testing.)

Some characteristics of our test programs are described in Table 1. For performance measurement we used gcc version 2.6.3 with optimization level -O4 to compile various versions of test programs. Execution times were measured on a dual processor Sun SPARCstation 20 running SunOS 5.4 (Solaris 2). This machine has two 66 Mhz Ross HyperSPARC with 256K L2 cache and 64MB of memory.

The core of our implementation is a source-to-source translator that generates a guarded program for an input C program. We compared the performance of our implementation of

basic guarding with that of Purify which is a very popular commercial tool modifying object files for a variety of run-time checks. See Section 2.5 for more details on Purify.

Adding code for run-time checking increases static program sizes as shown in Table 2. In purified programs the increase in text segment size is proportional to the static number of loads and stores, in guarded programs it is proportional to the static number of pointer operations. We do not know the implementation details of Purify but our best guess is that

Table 2: Guarding: Increase in program sizes

Program	Original (# of bytes in thousands)		Purified (% increase)		Guarded (% increase)	
	text	data+bss	text	data+bss	text	data+bss
alvinn	144.6	463.6	181.2%	9.9%	8.1%	0.2%
compress	103.7	424.7	218.8%	10.8%	12.9%	0.2%
ear	181.7	24.1	156.8%	190.9%	52.6%	7.5%
eqntott	121.2	284.7	194.6%	16.1%	111.7%	467.0%
espresso	274.2	15.2	131.5%	302.6%	355.5%	32.9%
sc	253.4	97.4	133.9%	47.1%	145.3%	313.4%
xlisp	197.4	13.8	151.1%	333.3%	191.5%	65.2%

the increase in data and bss segment sizes in purified programs is due to some internal data structures that a purified program maintains. For a guarded program, the increase is proportional to the global pointer declarations in the original program since they lead to declarations of corresponding guards (of size 5 times the original pointer size) in the guarded program. The increase is very pronounced in case of *eqntott* and *sc*. Both these programs

use parsers generated by the parser generator *yacc*. These parsers contain large global arrays of structures containing pointers generating even larger arrays of objects containing guards in the guarded program.

Run-time checking incurs overhead in execution time as shown in Table 3. The increase

Table 3: Guarding: Increase in execution time (user + system)

Program	Original (time in seconds)	Purified (% increase)	Guarded (% increase)
alvinn (50 epochs)	25.6 s	774.6%	472.3%
compress	2.9 s	748.3%	113.8%
decompress	2.0 s	765.0%	145.0%
ear	307.7 s	589.1%	642.0%
eqntott	18.8 s	735.6%	1337.2%
espresso	7.9 s	1107.6%	648.1%
sc (loadc2)	38.6 s	613.5%	173.3%
xlisp	122.1 s	988.1%	775.6%

in execution time for purified programs is proportional to the dynamic number of loads and stores in the original program. For guarded programs the increase is proportional to the dynamic number of pointer operations including array references. The pointer operations include pointer arithmetic, pointer copying, passing pointer parameters in addition to

pointer dereferences. The effect of having a large number of dynamic pointer operations is most apparent in the increased execution time for *eqntott*.

Run-time checking also increases the dynamic memory requirements of programs. Most of the dynamic memory a purified program uses is obtained using a call to `mmap()`, whereas guarded programs use heap-allocated dynamic memory. We could not meaningfully compare the increase in dynamic memory requirements for purified and guarded programs because it is very hard to monitor the amount of `mmap`ed dynamic memory on Solaris 2. The increase in heap usage for guarded programs is due to two factors: 1) dynamic allocations of structures/unions containing pointers and 2) sizes of the shadow heap and shadow stack. A dynamic allocation of structures/unions containing pointers leads to a larger allocation of objects containing guards. We maintain the auxiliary data structures 'shadow heap' and 'shadow stack' in the heap at run-time in guarded programs. The size of the shadow heap is proportional to the total number of dynamic allocations in the original program. The size of the shadow stack is proportional to the maximum depth of the run-time stack of the original program.

We found that the time to generate a guarded program is generally higher (up to 5-6 times in the worst case) than the time to generate a purified program. Increase in compilation time due to Purify's object level instrumentation can be high if an entire library has to be instrumented (as is the case with `libcurses` used in *sc*). Guarding increases processing time in two ways. First, the source-to-source translation with extensive analysis of

pointer declarations and uses can be time consuming; our implementation works in two phases (simplification followed by translation) paying the cost of disk I/O for simplified files. Second, the guarded program takes much longer to compile and link than the original program because of the instrumentation added.

2.5 Related Work

CodeCenter[17] is a programming environment that supports an interpreter-based development scheme for the C language. The evaluator in *CodeCenter* provides a wide range of run-time checks. It detects approximately 70 run-time violations involving illegal array and pointer accesses, improper function arguments, type mismatches etc. Interpretation of the intermediate code for supporting these checks is very expensive though; the evaluator executes C code approximately 200 times slower than the compiled object code.

Purify [11] is a commercially available system that modifies object files to, essentially implement a byte-level tagged architecture in software. It maintains a table at run-time to hold a two-bit state code for each byte in the memory. A byte can have a status of i) unallocated, ii) allocated but uninitialized, or iii) allocated & initialized. A call to a checking function is inserted before each load and store instruction in the input object files. This checking function verifies that the locations from which values are being loaded are readable (*i.e.* allocated and initialized) and the locations in which values are being stored are writable (*i.e.* allocated). Slowdowns by a factor of 5-6 are very common for Purified pointer intensive programs. Purify is very convenient to use because it works on object

files and can handle third-party libraries for which source code may not be readily available. However the major disadvantage of working at the object level is that Purify can not track the intended referents of pointers. Any access to memory that is in an allocated state is allowed. This severely restricts the kinds of errors that Purify detects. For example, an out of bounds array access can go undetected if it accesses a location belonging to another variable. If a pointer's intended referent is freed and the memory is reallocated, dereferencing the pointer should lead to a temporal access error; however Purify is also unable to detect that error. However, Purify detects more types of errors than our system (e.g. detecting un-initialized memory read) and also performs memory leak detection.

Austin *et al* [2] have proposed translation of C programs to *SafeC* form to handle array and pointer access errors. Their technique provides "complete" error detection under certain conditions. They have reported execution time overhead in the range of 130% to 540% for 6 (optimized) test programs. Their experimental system requires the user to convert each pointer to a *safe pointer* using a set of macros. A safe pointer is a structure containing the value of the original pointer and a number of *object attributes*. An input C program, annotated with macros, results in a C++ program which combined with some run-time support performs pointer access checking. Guarding shares the "completeness" of error detection with *SafeC*. Unlike the SafeC system, insertion of checks in our system is completely automated. Temporal access errors in SafeC are caught using a "capability" attribute which is an essentially unique value per object, much like the `lock` in shadow guards. However, checking temporal validity of a pointer access involves an expensive

associative search in a capability database. Such a search is avoided in shadow guarding by adding the `key_ptr` field in guards. The value of the `key_ptr` in shadow guards also serves to determine the storage class of objects. Hence a separate "storage class" attribute, as in safe pointers, to catch freeing of global objects is not necessary.

We are aware of commercially available memory access checking tools called *BoundsChecker* from NuMega Technologies (*http://www.numega.com/*) and *Insure++* from ParaSoft (*http://www.parsoft.com/*). Presumably these tools also work at the source level like our system. Unfortunately we could not meaningfully compare our system with these tools as we have not come across any publications describing the internal details of these tools.

Chapter 3

Customization

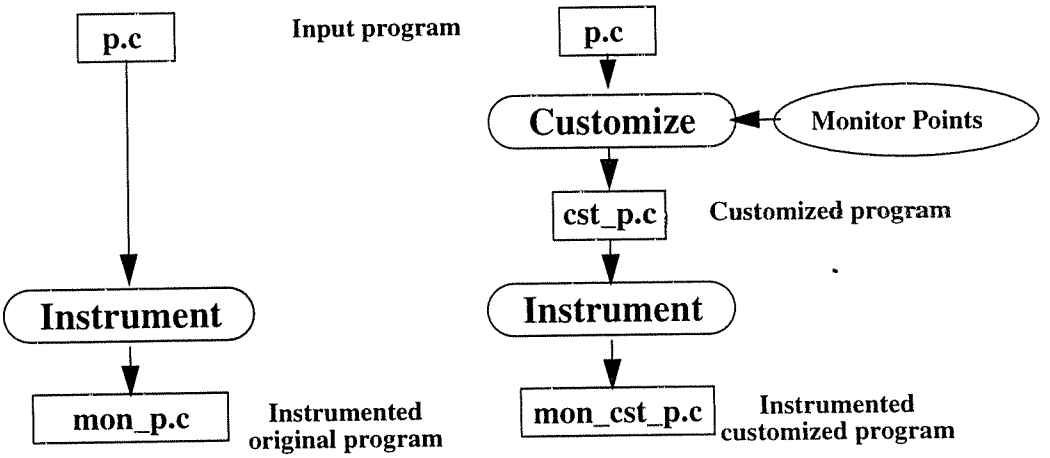
3.1 Introduction

Monitoring any activity in a program involves instrumenting the program to collect data or trigger run-time actions. An instrumentation tool first identifies points in the program where the activity of interest may be taking place — we call these program points the *monitor points*. The next step is to add code around the monitor points to actually collect data or trigger actions. For example, for producing branch statistics the monitor points include all the branches and function calls in the program; instrumentation is then added to increment counters.

For many activities of interest, instrumentation often slows down a program 3-4 times. Hence programs are not routinely monitored. Many researchers have focussed on using compile time analysis to reduce the cost of instrumentation. For example QPT, a profiling

and tracing tool[3], places code for incrementing branch counters at strategic places instead of at all the branch points. Other examples of static analysis include eliminating redundant array access checks[9] and efficiently deciding which variables to trace for data race detection[23].

We propose to speed up the monitoring process by decoupling monitoring from the original computation. Our approach is motivated by the observation that in practice an instrumented program is executed with the sole purpose of monitoring some activity — the results of the computation from an instrumented program are seldom of primary interest. The basic idea is to obtain a modified version of the original program that performs just enough computation to monitor the activity of interest. We call this modified version of the program a *customized program*. The customized program can then be instrumented to perform the desired monitoring. Figure 4 outlines two ways of instrumenting programs for



A: Traditional Approach B: Our Approach

Figure 4: Generating Monitoring Programs

monitoring. Figure 4:A shows the traditional approach where many optimizations are performed in the instrumentation phase. Figure 4:B outlines our approach to generating programs. Instead of focusing on how much instrumentation needs to be *included* for the given monitoring activity we focus on how much computation can be *excluded* from an instrumented program. The result is an input program customized for a given monitoring activity. The customized program is smaller and faster than the original program. It can then be instrumented for actual monitoring. Use of our technique is orthogonal to use of other static analysis techniques for reducing instrumentation overhead — all those techniques can be used during instrumentation of the customized program.

During this work, we experimented with customization in two different contexts 1) customization of shared-memory parallel C programs (SPLASH benchmarks [31]) for monitoring shared-memory accesses and synchronization operations and 2) customization of sequential C programs for guarding. We mainly used static program slicing for customization. A *program slice* consists of all the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. To customize a program for a given monitoring activity, we use the monitor points as the criteria for slicing the program.

Our customizer uses a slicing tool based on a slicing back end from the Wisconsin Program Integration System (WPIS). The back end operates on a program representation [14] called the system dependence graph (SDG). The algorithm described in [15] is used

to produce an interprocedural executable slice from the control flow and call graph information produced by a C front end.

The overall benefit of customization in terms of the reduction in monitoring overhead depends on three factors: 1) the frequency of occurrence of the monitor points, 2) the nature of the instrumentation added, and 3) the precision of the slicing tool used for customization. These factors are discussed in more detail in the sub-sections on performance later in this chapter.

3.2 Background and Related Work

The *slice* of a program with respect to program point p and variable x consists of all statements and predicates of that program that might affect the value of x at point p. An *executable slice* (version 2 of the slicing problems described in [14]) of a program with respect to program point p and variable x consists of a reduced program that computes the same sequence of values for x at p. One classification of program slices considers whether the slice is specific to a particular execution of the program for a given input (a *dynamic* slice) or the slice covers all possible executions of the program (a *static* slice). We are interested in executable static slices.

Tip [33] reports many applications of static and dynamic slices. Not many applications actually try to execute a *static* slice. One such application is proposed by Weiser in [34] where he describes use of executable static slicing for automatically parallelizing the execution of a sequential program. Several slices of a program are executed in parallel and the

outputs of the slices are spliced together in such a way that the input/output behavior of the original program is preserved. The splicing process may take place in parallel with the execution of slices. Only programs with structured control flow are considered.

The idea of executing modified, abstract versions of programs to reduce time and space overhead of tracing was used by a system called AE[20]. AE implements a technique called *Abstract Execution* for tracing incidents during a program's execution. Instead of instrumenting a program P to record a set of events E, Abstract Execution instruments P to collect a smaller set of significant events SE and also derives a program P' that uses SE to re-generate E. The implementation of abstract execution in the system AE uses a restricted form of *intra*-procedural slice to determine which events are significant. An event is considered significant if a backward slice with respect to that event includes a so-called *impossible instruction* (such as a function call). In addition, branches are always considered significant and the outcome of every branch condition is always recorded. P' uses SE, outcomes of branches and performs some recomputation to regenerate E. Our technique can be viewed as one extreme form of abstract execution where the set of significant events, SE, is empty[1]; and P' recomputes all the information, including the branch conditions, needed to regenerate E. Our technique uses a general, *inter*-procedural slicing algorithm to generate P'.

1.Except possibly the return values of system calls; see Section 3.5 for more details.

3.3 Customizing SPLASH benchmarks

We customized some programs from the SPLASH benchmarks [31] for monitoring synchronization operations and shared memory accesses in parallel programs. These monitor points are needed for tracing shared-memory programs for instant replay. The same monitor points can also be used for data race detection [5].

SPLASH programs are parallel programs with a shared memory programming model; they use explicit synchronization and process creation operations. Some characteristics of our test programs are described in Table 4.

Table 4: Characteristics of SPLASH programs tested

Program	# of files	# of procedures in the call graph	# of lines (preprocessed & simplified)
mp3d	3	44	4204
cholesky	9	74	5693
water	12	33	7876
locus	15	135	12105
pthor	25	258	29175

To generate a SPLASH program customized for execution replay we provided all the synchronization operations and all the shared memory references in the program as slicing criteria to our slicing tool. To side-step the alias problem while identifying the shared references, we modified the SPLASH benchmarks to make all the shared references explicit by converting them to macros. This is not unusual; the question of what shared memory

addresses are accessed by a statement is very hard to answer precisely (particularly for C programs). Many researchers either bypass the problem or make conservative assumptions [12]. Since our current approach of manually annotating shared references is very tedious and error-prone an automated analysis is definitely warranted.

3.3.1 Results

We used two criteria to determine the accuracy of the slices that are produced by our technique: 1) The slice should execute without any run-time errors. 2) The count of monitor points produced by executing the slice should match the count from the original program. A trivial instrumentation phase was built into our slicer that instruments the sliced programs to count monitor points.

Slicing and experiments were performed on a dual processor Sun SPARCstation 20 running SunOS 5.4 (Solaris 2). This machine has two 66 Mhz Ross HyperSPARC with 256K L2 cache and 64MB of memory. For performance measurement we used gcc version 2.6.3 with optimization level -O4 to compile various versions of test programs. We sliced the test programs using two kinds of slicing criteria 1) synchronization operations 2) synchronization operations and shared memory accesses. The time spent in the slicing back end

was a typically a few minutes, with the time for *pthor* being the maximum, around 30 minutes.

Table 5: Slicing SPLASH benchmarks: # of SDG vertices

Program	# of nodes in the system dependence graph (SDG)		
	Original	Slice: Synchronization only (% deleted)	Slice: Synchronizations & shared memory references (% deleted)
mp3d	5331	3831(**28.1%**)	4117 (**22.8%**)
cholesky	5680	3932 (**30.8%**)	4008(**29.4%**)
water	6411	5125 (**20.1%**)	5194 (**19.0%**)
locus	23909	15220 (**36.3%**)	17159(**28.2%**)
pthor	51289	21906(**57.3%**)	27844 (**45.7%**)

Looking at the static information (# of SDG nodes) in Table 5 as much as 1/2 of the system dependence graph gets "sliced away." This may not directly translate into an equivalent reduction in monitoring overhead because part of the SDG nodes deleted represent dead code which would not have been executed in the original program anyway. However, eliminating that dead code leads to smaller programs which are easier to analyze than the original program.

It appears that slicing may not benefit the smaller test programs very much because they are short, tightly written kernel routines with very little output — they are unlikely to have much computation that does not contribute to synchronization operations and shared memory references. Results for the larger applications are very encouraging. One interest-

ing point from the results in Table 5 is the that number of SDG nodes deleted for synchronization only is not much larger than the number for both synchronization and shared references. This may be due to the fact that most of the synchronization operations themselves use shared memory locations, hence most of the computations affecting shared references get included in a synchronization-only slice.

The real measure of how well our approach of generating monitoring programs compares with the traditional approach (Figure 4) is the ratio of the execution time of the customized monitoring program (**mon_cst_p.c** in Figure 4:B) to the execution time of a monitoring program obtained using the traditional approach(**mon_p.c** in Figure 4). However, this ratio also depends on the nature of actual instrumentation and optimizations performed during instrumentation. Unfortunately, instrumenting SPLASH benchmarks for any real task (e.g. race detection) was beyond the scope of this work. To judge the potential benefit of our slicing approach in isolation, we instrumented the SPLASH programs to simply count the number of synchronizations and shared memory accesses. Table 6 and Table 7 present the overhead of "trivial" monitoring of SPLASH programs. The instruc-

tion counts reported in these tables were obtained using the utilities *qpt2* and *qpt2_stats* [3].

Table 6: Effect of Customization: Counting Synchronizations

Program	Original (dynamic instruction count in millions)	Frequency of synchronizations [once every N instructions]	% Overhead	
			Traditional approach	Our approach
mp3d	17.9	31470	0.01%	-0.2%
cholesky	34.0	689	0.6%	0.2%
water	28.4	2270	0.1%	-0.0%
locus	815.0	10008	0.1%	-4.7%
pthor	76.9	2107	0.2%	-2.4%

The monitor points used for customizing the SPLASH programs in our experiments are very sparse. This is indicated by the numbers in the third columns of Table 6 and Table 7. Since the monitor points are executed so infrequently the basic overhead of counting the monitor points with the traditional approach is not much; however customization with slicing reduces this overhead even further. In some cases the customized program actually ran faster than the original program.

Since the instrumentation added is extremely light weight we believe the improvements obtained for "trivial" instrumentation give an upper bound on the benefit of customization. On the other hand, the numbers presented in Table 6 and Table 7 can be further improved with a better alias detection algorithm in our slicing tool as currently many conservative

assumptions about possible side-effects of pointer dereferences result in larger program slices than are needed.

Table 7: Effect of Customization: Counting Synchronizations and Shared accesses

Program	Original (dynamic instruction count in millions)	Frequency of synchronizations and shared accesses [once every N instructions]	% Overhead	
			Traditional approach	Our approach
mp3d	17.9	1483	0.2%	0.1%
cholesky	34.0	689	0.6%	0.2%
water	28.4	1133	0.2%	0.1%
locus	815.0	9	32.6%	28.8%
pthor	76.9	200	1.6%	-0.1%

3.4 Customization for guarding

Checking pointer and array accesses is an expensive operation; it routinely slows down the input program 3-4 times. Most past attempts at reducing this overhead have concentrated on using compile time analysis to reduce the number of run-time checking assertions. There has been much work on eliminating redundant array bound checks in FORTRAN [9, 18]. Unfortunately techniques for FORTRAN can not be readily applied to C programs because of the presence of pointers. Pointer arithmetic, equivalence of arrays and pointers, and aliasing in C programs require conservative assumptions to be made during compile-time analysis for eliminating run-time checks. We have implemented such a

conservative analysis in our guarding prototype. We were able to remove up to 10% of the run-time checks from test programs.

As an alternative way to reduce the guarding overhead we experimented with customizing C programs for pointer and array access checking. As outlined in Figure 4, we use the pointer and array accesses as the monitor points for customizing an input program (`p.c`) for guarding. The reduced, customized program (**cst_p.c**) can then be instrumented for run-time checking. We call the instrumented reduced program (**mon_cst_p.c**) a *customized guarded program.*

We first implemented a simple scheme to generate a reduced program for guarding. We look at calls in the user program to functions not defined by the user. These include calls to external functions — library calls and system calls. We term calls to functions such as **printf** which affect the external environment as "output calls." Some output calls such as writing into a file must not be repeated in the guarded program lest they interfere with the result of the user program. Most output calls clearly do not affect the pointer and array operations in the user program. We have modified the translator for basic guarding shown in Figure 1 for deleting output calls. We maintain a database of external calls. For each external function the database stores the category of the call — whether the call is an "output call" or not. Our modified translator consults this database to delete the output calls from the user program and adds code for maintaining and checking guards to generate a customized guarded program. Deleting output calls in turn results in deletion of some

more computation (via dead code elimination) contributing solely to those calls. We have found that this basic technique of deleting output calls can result in a customized guarded program that runs up to 23% faster than an embedded checking program (non-customized guarded program).

We found some test cases in which the customized guarded program obtained by deleting output calls takes almost as much time as the standard embedded checking approach. These programs do not spend much time in the output routines, hence the value of reducing output calls is minor.

The speed of guarding can be further improved by using slicing technology to remove computations not necessary for guarding. To customize the user program for guarding, we slice the user program using the pointer and array accesses as the criteria for slicing. The result is a smaller and faster executable program that performs only the computations affecting the array indices and pointers being dereferenced. This program is then instrumented to generate a guarded program. A customized guarded program obtained using slicing can run more than twice as fast as an embedded checking program.

Our translator for generating a customized guarded program using slicing works in 4 phases. The first phase processes simplified files one-by-one. For each simplified file it produces two files: 1) a file containing the control flow graph (CFG) for the code in the simplified file along with node numbers for the abstract syntax tree (AST) and 2) another file containing the slicing criteria in terms of AST node numbers for array accesses and

pointer dereferences in the simplified file. The second phase of our translator reads in the pairs of files for all the simplified files produced by the first phase and calls a slicing routine in the slicing back end with nodes in the CFG marked according to the slicing criteria. The slicing back end gives back results by marking the CFG nodes in the slice. For each simplified file, phase two of our translator then produces a file containing numbers of AST nodes in the resulting slice. Phase three processes simplified files one by one using the corresponding slicing results file from phase two and generates sliced simplified files. Phase four is exactly like the original translator in Figure 1, except that it reads in a *sliced* simplified file. The result is a customized guarded program obtained using slicing.

3.4.1 Performance of Guarding with Customization

To judge the effectiveness of various overhead reduction techniques using customization we compared the performance of customized guarded programs with those of basic, non-customized guarded programs.Table 8 presents execution times (user and system) for a few of our test cases with and without output calls in the guarded program. Deleting output calls mainly reduces system time; there is also some change in the user time due to elimination of code rendered dead by deleting output calls. We noticed that deleting output calls did affect all the programs we tested; the effect is most pronounced for programs with a lot of output calls at run-time. An example of such a program is *queens* in Table 8

where deleting output calls made the customized guarded program run not only faster than

Table 8: Customized guarding: Effect of deleting output calls

user and system times					
Program		**com-press**	**decom-press**	**ear**	**queens (-a 11)**
Original	user	2.3 s	1.3 s	302.5 s	1.0 s
	system	0.6 s	0.7 s	5.2 s	3.5 s
	total	2.9 s	2.0 s	307.7 s	4.5 s
Guarded	user	5.2 s	4.0 s	2277.9 s	3.7 s
	system	1.0 s	0.9 s	5.3 s	3.5 s
	total	6.2 s	4.9 s	2283.2 s	7.2 s
Customized Guarded (Without output calls)	user	4.8 s	3.4 s	2264.0 s	2.9 s
	system	0.4 s	0.2 s	0.5 s	0.1 s
	total	5.2 s	3.6 s	2264.5 s	3.0 s

the basic guarded program but also faster than the original program. This test program (from McGill University) prints all the solutions to the 11 queens problem. The original program and the basic guarded program spend a lot of time in output calls, most of which vanishes from the customized guarded program. We believe that *queens* is representative of a certain class of programs such as graphics utilities that spend a lot of time in the output calls and can benefit greatly from customized guarding by deletion of output calls.

We did notice that for some of our test cases deleting output calls from the guarded program has hardly any effect on execution time. We are have experimented with using slicing technology to customize programs as outlined in Section 3.3. We have been able to slice some of our test cases to obtain sliced customized guarded programs. Table 9 presents the effect of customization on reducing the overhead of "trivial" monitoring of deref-

Table 9: Effect of Customization: Counting Dereferences

Program	Original (dynamic instruction count in millions)	Frequency of dereferences [once every N instructions]	% Overhead	
			Traditional approach	Our approach
alvinn (50 epochs)	1224.6	3	82.0%	-39.0%
compress	92.0	11	24.8%	6.8%
decompress	76.0	13	16.9%	-24.9%
queens (-a 11)	112.2	21	14.2%	-50.2%
stanford (20 iterations)	447.3	5	55.2%	49.3%

erences in some test cases. The dereferences occur rather frequently in these cases as indicated by the numbers in column two of Table 9. Correspondingly, the basic overhead of counting dereferences with the traditional approach is high. Customization reduced the overhead of counting dereferences in all the cases; in some cases the customized program actually ran faster than the original program.

We also instrumented our customized test programs for guarding. The results are presented in Table 10. This instrumentation is much more heavy-weight than the "trivial"

Table 10: Effect of Customization: Guarding

Program	% Overhead (dynamic instruction counts)	
	Traditional approach	**Our approach**
alvinn (50 epochs)	635.4%	328.7%
compress	212.8%	193.6%
decompress	223.2%	173.6%
queens (-a 11)	280.0%	141.7%
stanford (20 iterations)	568.2%	560.0%

instrumentation for counting. Further, unlike in the "trivial" monitoring instrumentation is added not only at the dereferences but also at various other pointer operations. Hence the basic monitoring overhead using the traditional approach is very high. Customization reduced this overhead in all the test cases.

Table 11 presents the execution time overhead for the purified programs and 3 versions

Table 11: Customized guarding: Effect of slicing

Execution time (user + system)					
Program	**Original (time in seconds)**	**Purified (% increase)**	**Guarded (% increase)**	**Customized Guarded (no output) (% increase)**	**Customized Guarded (sliced) (% increase)**
alvinn (50 epochs)	25.1 s	792.0%	483.7%	482.9%	282.1%
compress	2.9 s	748.3%	113.8%	79.3%	10.3%
decompress	2.0 s	765.0%	145.0%	80.0%	45.0%
queens (-a 11)	4.5 s	233.3%	60.0%	-33.3%	-60.0%
stanford (20 iterations)	10.4 s	659.6%	430.8%	426.0%	239.4%

of guarded program. Using slicing to customize guarding looks very promising; it subsumes customization by deleting output calls. Slicing helped us to reduce the execution time overhead of guarding for all the test programs we were able to slice. The version of the slicing tool used to generate the customized guarding programs for the numbers reported in Table 11 is different from the tool used for other tables in this chapter. This version of the tool made more aggressive assumptions about the side-effects of library functions. Consequently, the overhead reduction with customized guarding is more pronounced in Table 11 than in Table 10. Another difference between the two tables is that

the former presents increase in dynamic instruction counts while the latter presents increase in (user + system) time.

3.5 Discussion

Calls to operating system kernel routines (system calls) is one source of non-determinism for both sequential and parallel programs. A system call is a well defined entry point into operating system code. The return value of a system call depends on the state of the operating system data structures at the time the call was made. If we want the monitored program to get the exact same return values from the system calls as the original program then we will have to instrument the original program to record the return values of system calls. From our experience with SPEC92 benchmarks, the number of values to be recorded are too few to cause any significant difference in the execution time of the original program. We believe that letting the monitored program repeat the system calls is safe —as it will monitor *some feasible* run of the original program. It allows us to run a monitored program independent of the original program.

An interesting question to ask is when to run the monitoring program. There are two alternatives: 1) Concurrently with the original program, 2) On its own — post/pre execution.

The first alternative requires extra processors; but can generate monitoring information concurrently with the results of the original program. An extra processor can be used to *routinely* monitor programs in the background. This should work well for sequential pro-

grams as shared memory multiprocessors are becoming increasingly common. We have used idle processors in a multiprocessor workstation to perform memory access checking in the background using a technique called *shadow processing*. Shadow processing is described in detail in Chapter 4.

Alternative 2 (pre or post execution) may be more practical for parallel programs since users typically want to use all the parallelism available. However, the speedup obtained for most parallel programs is far from linear. The incremental benefit of allocating extra processors to a parallel program starts diminishing. At some point it may make sense to dedicate a couple of processors for executing a monitoring program.

Chapter 4

Shadow Processing

4.1 Introduction

In this chapter we present a technique called *shadow processing* that uses idle processors in a multiprocessor workstation to perform concurrent monitoring. We have implemented a system for concurrent memory access checking using shadow processing. We create two processes for a user program. One is the *main* process, executing the user program as usual, without run-time checking. The other is a *shadow* process, following the main process and verifying the legality of the array and pointer accesses. The shadow process maintains guards for the pointers and arrays in the main process. The major advantage is that since memory access checking is taken out of the critical execution path of the main program; the overhead to the main process (which the user sees) is less than 10% in most

cases. Thus there is no need to turn off run-time memory access checking; every execution of the main process can be monitored by a shadow process in the background.

4.2 Motivation

Use of multiprocessors is no longer restricted to big corporations and research institutes. Low-end bus based shared memory multiprocessors are widely available today. Vendors such as Sun and SGI offer multiprocessor workstations. Dual processor PCs have started appearing in the market. With rapid advances in microprocessor technology, high-performance microprocessors should soon routinely incorporate as many as four general-purpose central processing units on a single chip [8].

The goal, of course, is to use extra processors in a multi processor workstation to solve problems beyond the scope of a single processor. The reality is that such a workstation often operates like a single-user time-sharing system. The Computer Sciences Department of the University of Wisconsin—Madison has more than 60 dual processor workstations. We monitored 18 of those machines over a period of 24 hours using 'condor'[21]. We found that 83% of the dual processor workstations that were active had a load average less than one. This implies that most of the time a dual processor workstation was busy, one processor was running an application while the other processor was idle. We plan to use idle processors in multiprocessor workstations to perform a variety of useful tasks using *shadow processing*.

4.3 Concurrent monitoring using Shadow Processing

The basic idea in shadow processing is to partition an executable program into two run-time processes as shown in Figure 5. One is the **main process**, executing as usual. The

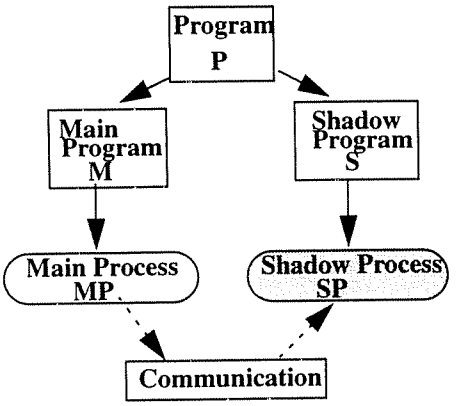


Figure 5: Shadow Processing

other is a **shadow process**, following the main process and performing auxiliary tasks such as run-time checking or profiling. The two processes may communicate and synchronize during execution. The model of shadow processing in Figure 5 is very general; the input program may be in any language. Further, there are no assumptions about the communication and synchronization mechanisms in the underlying multiprocessor architecture (shared memory or message passing).

A shadow program is an abstract version of the main program, executing only those statements that are relevant for the activity being monitored. Since the shadow program executes in the background, not interfering with the user, run-time analyses that normally seem too expensive become attractive with shadow processing. Typical applications include checking validity of memory accesses, executing user defined assertions, reporting

side effects of functions, determining coverage of test suites, detecting memory leaks, and performing tag-free garbage collection of strongly typed languages.

The control flow of the shadow is determined by its own computations and by some crucial values communicated by the main process. Let us look at two extreme ways of determining the control flow of a shadow process performing some run-time analysis. First, imagine the shadow program to be an exact copy of the main program with statements added for the analysis. In this case the two processes need only communicate certain values that can not be safely recomputed in the shadow (e.g. interactive input, system calls etc.). Generating the main and the shadow program in this case involves identifying the points in the input program requiring communication. The extra work to be done by the main process at run-time is minimal. But the shadow repeats all the computations of the main process, many of which may not be necessary for the analysis being performed. At the other extreme, the main process communicates every control decision to the shadow process. This clearly puts a lot of overhead on the main process. However, the shadow program in this case is very easy to generate — it is merely a control flow skeleton of the main program with analysis-specific statements.

The first alternative described above minimizes overhead to the main process but results of the run-time analysis are delayed. The second alternative tries to speed up the run-time analysis performed in the shadow process but with increased overhead to the main process. A continuum exists between these two extremes. The ideal case is when the

main process is burdened by only the communication of irreproducible values and the shadow performs only those computations that are necessary for the application at hand. This approach will truly utilize the benefit of parallelism, optimizing the overall time required to perform the run-time analysis. The shadow program in this case is obtained from the main program by extracting all those statements that affect the analysis being performed. This can be achieved by taking a program slice[33] with the slicing criteria involving the variables required for the analysis. These slices can be combined to form the shadow program. The technique of program customization presented in Chapter 3 can be readily used to generate a shadow program; some extra instrumentation may have to be added to a customized program if communication of any values is desired.

4.3.1 Shadow run-time checking

The high cost of run-time checks restricts their use to the program development phase. When programs are fully developed and tested, they are assumed to be correct and run-time checks are disabled. This is dangerous because errors in heavily-used programs can be extremely destructive. They may not always manifest themselves as a program crash but may instead produce a subtly wrong answer. Even if an erroneous program crashes, it may be difficult to repeat the error inside a debugger. Further, debugging long running programs can be very time consuming. Undiscovered errors in heavily-used programs may not be rare; a study [24] has shown that as many as a quarter of the most commonly used Unix utilities crash or hang when presented with unexpected inputs. Thus there is a strong

case for running programs with checks routinely enabled. Naturally, these checks should be as inexpensive as possible.

Envision the shadow process as a copy of the main program with all the desired run-time checks added. Communication is minimal — it is needed only for those values that cannot be safely recomputed by the shadow process (e.g. interactive input, return values of system calls etc.). A shadow checking process repeating all the computations in the main program will definitely run slower than the main process, but this may well be acceptable if errors need not be detected at the exact microsecond they occur. Furthermore, with a careful analysis, the shadow process need not reproduce the main process' full computation, but rather only those values that need to be monitored. Hence a shadow process need not greatly lag behind the main process; it may be able to detect errors in almost real time.

A major plus in executing a checking program on an extra processor is that an error report is available concurrently with the results of the original program. However the main and the shadow processes share the system resources such as memory. It is possible that for some memory-hungry programs the memory requirement of the shadow process may result in page thrashing, slowing down the main process. Also, if the shadow needs to exactly follow the execution path of the main process, certain irreproducible values such as interactive input and return values of system calls need to be read from the main process. This communication may further slow down the main process. We have modified our translator for shadow processing to consult a database to decide whether communication

is needed on a particular external call. This database needs to be updated manually if new external calls are encountered. The translator adds code to the main and shadow process to communicate values using a circular buffer in shared memory (implemented using a system call `mmap()`). As stated in the section on guarding, we believe that it is safe to let the guarded program (running on its own or concurrently with the original program) repeat all the external function calls including the system calls.

Dedicating a processor for a shadow process may not always be feasible on heavily used multiprocessor workstations. For uniprocessor systems and heavily used multiprocessor systems the alternative of running the shadow program in isolation as resources become available is more practical.

4.4 Performance of Shadow Guarding

Table 12 summarizes the execution time for some SPEC92 benchmarks for concurrent guarding using shadow processing. The overhead to the main process is mainly due to memory conflicts with the shadow process. In the case of *sc*, communication of return values of library calls also contributes to the overhead; there was no communication for the other test cases. The overhead indicates the delay in obtaining results of the original computation in the shadow processing environment. It is below 10% in most of the cases. To

the average user, the main process will appear almost indistinguishable from the original un-instrumented program.

Table 12: Concurrent Guarding using Shadow Processing: (user + system) time

Program	Original (time in seconds)	Embedded Checking		Shadow Guarding	
		Purified (% increase)	Guarded (% increase)	Main (% increase)	Shadow: without output calls (% increase)
alvinn (50 epochs)	25.6 s	774.6%	472.3%	< 1%	472.3%
compress	2.9 s	748.3%	113.8%	10.3%	89.7%
decom-press	2.0 s	765.0%	145.0%	10.0%	85.0%
ear	305.4 s	589.1%	642.0%	< 1%	636.4%
eqntott	18.8 s	735.6%	1337.2%	3.2%	1328.2%
espresso	7.9 s	1107.6%	648.1%	1.3%	650.6%
sc (loadc2)	38.6 s	613.5%	173.3%	6.7%	95.1%
xlisp	122.1 s	988.1%	775.6%	5.6%	787.6%

4.5 Other concurrent dynamic analysis techniques

ANNA (Annotated ADA) is an Ada language extension that allows user defined executable assertions (checking code) about program behavior. An ANNA to ADA transformer that allows either sequential or concurrent execution of the checking code is described in [28]. Concurrent run-time monitoring is achieved by defining an ADA task containing a checking function for each annotation. Calls to the checking function are automatically

inserted at places where inconsistency with respect to the annotation can arise. Like shadow processing, the ANNA to ADA transformer uses the idea of executing checking code concurrently with the underlying program. However, it generates numerous tasks per annotation, which may lead to excessive overhead. Executing user defined assertions seems like a good application for shadow processing.

Parasight [1] is a parallel programming environment for shared-memory multiprocessors. The system allows creation of observer programs ("parasites") that run concurrently with a target program and monitor its behavior. Facilities to define instrumentation points ("scan-points") or "hooks" into a running target program and dynamically link user defined routines at those points are provided. Threads of control that communicate with the parasites using shared-memory can be spawned. Parasight is an interactive system geared towards debugging of programs. The overhead incurred in the target program because of "hooking in" of parasites is not an issue. Shadow processing can use some of the ideas from Parasight. In certain applications, the shadow process need not be active for the whole execution of the main program. It could be "hooked in" with an already executing main process when a processor becomes available, "spot checking" the main program. The shadow can start executing at certain well defined points, say at the entry of functions. The main process will have to leave a trail of indicators (in a shared buffer) indicating those points have been reached.

One approach to concurrent run-time checking is to use specialized hardware. Tagged hardware [6] can be used for type-checking at run-time. Watchdog processors [22] are used to provide control flow checking. The Makbilan architecture [27] has been proposed for non-intrusive monitoring of parallel programs in parallel. Unfortunately specialized architectures are not widely available and they may not be able to support the full range of desirable checks (e.g., pointer validity checking).

Chapter 5

Conclusions

5.1 Summary

In this thesis, two techniques to handle the high run-time costs of software-based program monitoring were presented 1) customization and 2) shadow processing. These techniques were mainly tested in the context of memory access checking of C programs using a technique called guarding.

Customization is a technique that can help speed up any monitoring activity such as memory access checking or profiling. The basic idea is to obtain a reduced version of a user program customized to the monitoring activity. We explored two ways to customize a user program for guarding. First by deleting calls to library routines affecting the external environment (output calls) and second by using a prototype program slicing tool to throw away irrelevant computations from the user program. Deleting output calls benefits a class

of programs performing a lot of graphics or terminal output the most. Slicing is a more general technique that can benefit a wider range of programs. We also used our prototype slicing tool to customize programs from the SPLASH benchmarks for monitoring synchronizations and shared memory access. We found that customization can double the speed of monitoring.

Shadow processing is a technique that uses idle processors in multiprocessor machines to perform monitoring. We have used shadow processing for concurrent guarding. Current approaches to pointer access checking work sequentially, typically slowing a computation 3-4 times. Such high overheads make those approaches unsuitable for heavily-used programs. After programs are fully developed and tested, running them with embedded checks seems unacceptably slow. Most programmers turn off the checks, trading reliability for speed. Shadow guarding offers an excellent way out – a shadow process works silently in the background watching for run-time errors. Computations in the user program are performed by a main process. Error-free runs of the main process are only slightly slower than the original. Occasional erroneous runs lead to an error report (sometimes slightly delayed) from the shadow process. If the original program crashes, an error report from the shadow points to the root cause of the crash. Reports on errors that do not crash the original program can be extremely helpful in uncovering hidden bugs.

5.2 Future Work

The guarding technique for C programs presented in this thesis can be easily adapted to languages other than C. It can also be extended to handle a wider range of errors. In [25] we presented a way to extend guarding for memory leak detection. However in our preliminary experiments we found the overhead of memory leak detection using guarding very high mainly because we relied solely on source-level instrumentation. Although source-level instrumentation of our guarding prototype results in better error-coverage for certain kinds of errors it also limits the applicability of guarding for third party software whose source code is not available. We believe a checking tool combining source-level instrumentation with object code insertion could be extremely powerful.

Customization using slicing in its most general form can be computationally expensive. However this high cost can be amortized over a large number of runs of the customized monitoring program. By giving the proper slicing criteria, monitoring only a selected part of the user program is possible.

One useful application of customization could be in the form of a tool for "bug-isolation." As programs get bigger debugging them gets harder. If a programmer is able to narrow a bug down to say a particular function; an executable slice with respect to that function could result in a program which manifests the original program but is smaller hence easier to debug than the original program.

Shadow processing uses idle processors in multiprocessor workstations to hide the overhead of monitoring. This amounts to exploiting "task level" parallelism between computation and monitoring. Another avenue for exploration is to utilize "instruction-level parallelism" to hide the instrumentation overhead. Many modern microprocessors make it possible to issue and execute a large number of instructions concurrently. Exploiting this parallelism effectively is quite a challenge to compiler writers[16]. In the absence of useful instructions to execute, a lot of processor cycles remain unused on many modern microprocessors. There have been attempts to exploit these unused "delay slots" for some useful work. Kurlander and Fischer [19] proposed using delay slots for "range-splitting" to improve code scheduling. Unused delay slots were used by Schnarr and Larus [29] to hide profiling overhead. Similarly, error-detection overhead could also be hidden by putting the instrumentation in delay slots.

The overhead reduction techniques of customization and shadow processing can be applied to many important monitoring activities. We believe our techniques show great potential in reducing the cost of software-based monitoring and hence making routine monitoring of production programs more affordable.

Bibliography

[1] Ziya Aral and Ilya Gertner. High-Level Debugging in Parasight. In *ACM Workshop on Parallel and Distributed Debugging*, pages 151–162, May 1988.

[2] Todd Austin, Scott Breach, and Gurindar Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[3] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[4] Brian Case. Updated SPEC Benchmarks Released. *Microprocessor Report*, pages 14–19, September 16, 1992.

[5] Anne Dinning and Edith Schonberg. Detecting Access Anomalies in Programs with Critical Sections. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):85–96, December 1991.

[6] E. A. Feustal. On the Advantages of Tagged Architectures. *IEEE Transactions on Computers*, C-22(7):1241–1258, July 1973.

[7] Charles N. Fischer and Richard J. LeBlanc Jr. Benjamin/Cummings Publishing Company Inc., 1991.

[8] Patrick Gelsinger, Paolo Gargini, Gerhard Parker, and Albert Y. C. Yu. Microprocessors circa 2000. *IEEE Spectrum*, pages 43–47, October 1989.

[9] Rajiv Gupta. Optimizing Array Bound Checks Using Flow Analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, March-December 1993.

[10] Samual P. Harbison and Guy L. Steele Jr. *C - A Reference Manual*. Prentice Hall, 3rd edition, 1991.

[11] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, 1992.

[12] D. P. Helmbold and C. E. McDowell. A taxonomy of race detection algorithms. Technical Report UCSC-CRL-94-35, University of California, Santa Cruz, Board of studies in Computer and Information Sciences, September 1994.

[13] Laurie J. Hendren and Bhama Sridharan. The SIMPLE AST - McCAT Compiler. ACAPS design note 36, School of Computer Science, McGill University, Montreal, Canada, 1992.

[14] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[15] Susan Horwitz, Thomas Reps, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *SIGSOFT'94:Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[16] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllnhaal, and D. I. August. Compiler Technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1625–1640, December 1995.

[17] Stephan Kaufer, Russel Lopez, and Sesha Pratap. Saber-C an Interpreter-based Programming Environment for the C Language. In *Proceedings of the Summer USENIX Conference*, pages 161–171, 1988.

[18] Priyadarshan Kolte and Micheal Wolfe. Elimination of Redundant Array Subscript Range Checks. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[19] Steven M. Kurlander and Charles N. Fischer. Zero-cost Range Splitting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[20] James Larus. Abstract Execution: A Technique for Efficiently Tracing Programs. *Software - Practice and Experience*, 20(12):1241–1258, December 1990.

[21] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[22] A. Mahmood and McCluskey E. J. Concurrent Error Detection using Watchdog Processor - A Survey. *IEEE Transactions on Computers*, C-37(2):160–174, February 1988.

[23] John Mellor-Crummey. Compile-time Support for Efficient Data Race Detection in Shared-Memory Parallel Programs. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 129–139, San Diego, California, May 1993.

[24] Barton P. Miller, Fredriksen Lars, and Brian So. An Empirical Study of the Reliability of Unix Utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[25] Harish Patil and Charles Fischer. Efficient Run-time Monitoring Using Shadow Processing. In *2nd International workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, St. Malo, France, May 1995.

[26] Thomas Reps. Demonstration of a prototype tool for program integration. TR 819, Computer Sciences department, University of Wisconsin, Madison, Wisconsin, January 1989.

[27] Robert Rubin, Larry Rudolph, and Dror Zernik. Debugging Parallel Programs in Parallel. In *ACM Workshop on Parallel and Distributed Debugging*, pages 216–225, May 1988.

[28] Sriram Sankar and Manas Mandal. Concurrent Runtime Monitoring of Formally Specified Programs. *Computer*, 26(3):32–41, March 1993.

[29] Eric Schnarr and James Larus. Instruction Scheduling and Executable Editing. In *Workshop on Compiler Support for System Software, (WCSSS'96)*, Tucson, Arizona, February 1996.

[30] Beth A. Schroeder. On-line monitoring: a tutorial. *Computer*, 28(6):72–78, June 1995.

[31] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[32] Joseph L. Steffen. Adding Run-time Checking to the Portable C Compiler. *Software - Practice and Experience*, 22(4):825–834, April 1992.

[33] Frank Tip. A Survey of Program Slicing Techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[34] M. Weiser. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters*, 17:129–135, October 1983.

Appendix:Bug Report from Guarding

We have used our system for pointer and array access checking to test programs from a variety of sources including SunOS 4.1.3 utilities, SPEC 92 benchmarks, and the SPLASH multiprocessor benchmarks. In the following we describe some of the errors we discovered. A utility called *fuzz* [24] was used to generate random input for the SunOs utilities. SPEC92 and SPLASH benchmarks were tested with their reference inputs, unless stated otherwise.

- **Benchmark:** decompress
- **Source:** SPEC92
- **Error #1:** file compress.c

Function `getcode()` called from `decompress()`

```
1144  /* high order bits */
1145  code |= (*bp & rmask[bits]) << r_off;
```

Pointer `bp` is used to traverse global character array `buf[BITS]`. While decompressing the reference input the number of bits per code changes to 16. (This condition can be forced by changing `#define INIT_BITS to 16`). Sometime after this happens, `bp` dereferences one location beyond the array `buf` at line 1145. This error can be verified by putting '`assert((bp-&buf[0])<BITS);`' before line 1145.

The purified program did not report any errors.

- **Benchmark:** sc
- **Source:** SPEC92
- **Error #1**: File `sc.c`, in function `update()`:

```
213   /* Now pick up the counts again */
214   for (i = stcol, cols = 0, col = RESCOL;
215         (col + fwidth[i]) < COLS-1 && i < MAXCOLS; i++) {
```

An array bound violation occurs in the terminating condition of the for loop.The two operands of `&&` are in the wrong order. `i < MAXCOLS` must come before indexing

`fwidth[i]`. When `i` becomes `MAXCOLS (40)` there is an array bound violation. (Occurs twice for input.ref/loada2)

- **Error #2:** Found error in file `lex.c`, in function `yylex()`:

```
114     for (tblp = linelim ? experres : statres; tblp->key; tblp++)
115             if (((tblp->key[0]^tokenst[0])&0137)==0
116              && tblp->key[tokenl]==0) {
```

Array `tokenst` contains the current token and `tokenl` is the length of the current token. The for loop traverses a table of reserved words to see if the current token is a reserved word. Pointer `tblp` is used to point to various entries of a table of reserved words.The first part of the if condition checks if the first letter of the current reserved word and the current token are the same. (It uses some clever bit manipulation and properties of the ASCII character set to ignore case differences.) The second part makes sure that the current reserved word is not longer than the current token. For input token `goto`, `tokenl` is `4`. There is a keyword `GET` in the table `statres`. The first part of the if condition is satisfied `(('G'^'g'&0137)==0)` is `TRUE`). For the second check, `tblp->key[tokenl]` accesses the 5th element of the current key `GET` which has length 4 (`'G','E','T','\0'`). Accessing the 5th element of an array of size 4 is clearly illegal.

The purified program misses these two errors. However, it reports 13 errors (uninitialized memory reads) in the libraries `libc` and `libcurses`.

- **Benchmark:** cholesky
- **Source:** SPLASH-1
- **Error #1:** file util.U

Function `ReadSparse5()` defines `char type[3];`

```
166  type[3] = 0
```

Only valid indices are `0-2`. This error has been fixed in the latest release of the SPLASH benchmarks (SPLASH-2). The purified program did not report any error.

- **Error #2:** file util.U
  Function `ISort()`:

  350 while (M.row[j-1] > tmp && j > lo){ ...
  On some calls to `ISort()` when the value of `lo` is `0`, index of `M.row[]` becomes `1`

which is illegal. The statement should have been:

  350 while (j > lo && M.row[j-1] > tmp){ ...
  This error is still there in the latest release of the SPLASH benchmarks — SPLASH-2.

The purified program did report this error and it also reported 9 uninitialized memory reads in the library function `sscanf()`.

- **Benchmark:** locus
- **Source:** SPLASH-1
- **Error #1:** file ginput.U
  Function `ReadSparse5()`:

  201 if (NumberOfTerminals < 2){
  ...
  203 free(NewWire);
  `free(NewWire)` is followed by a dereference of `NewWire`:

  222 NewWire->NumberOfGroups = IGrouptNumber;
  The purified program did report this error.

- **Error #2:** file timer.U
  Function `ReportTimes()`

  161 for(i=0; i <=Global->NumberOfWires; i++){
  ...
  162 if(SegmentUsed[i] != 0) && (NetPinDistribution[i] != 0)){ ...
  Valid indices for `SegmentUsed[]` and `NetPinDistribution[]` are `0-800`. For one of the reference inputs `Global->NumberOfWires` is `904` and invalid indices of the `SegmentUsed` and `NetPinDistribution` are accessed. The purified program did report this error and it also reported one uninitialized memory read.

  "locus" is not part of SPLASH-2.

- **Utility:** col
- **Source:** SunOs 4.1.3
- **Error #1:** file col.c

  229 c3 = *line;

Inside the program, there are many places where the pointer `line` is incremented and dereferenced without checking its validity.

The purified program does catch this error for some inputs. We have found a random input for which the purified program does not terminate (neither does the original program).

- **Utility:** ul
- **Source:** SunOs 4.1.3
- **Error #1:** file ul.c

There are numerous array bound violations in function `filter()`. Array `obuf` is indexed using the variable `col`. If an input line contains >=512 characters `col` exceeds the maximum index (511) allowed for `obuf`.

The purified program does not catch this error for some inputs. We have constructed a sample input for which the purified program does not report this error.