Shadow Guarding*: Run-time Checking You Can Afford

Harish Patil and Charles Fischer
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{patil, fischer}@cs.wisc.edu

November 9, 1994

Abstract

Shared memory multiprocessors are becoming increasingly common. We present a technique that uses shared memory multiprocessors to efficiently implement run-time checking. The user program pays a very small premium on each run. In return there is an assurance of reliability. A second "shadow" process looks for errors in the background. For the error-free runs, the user process hardly notices the shadow process; for erroneous runs, the shadow process provides useful error information automatically.

1 Introduction

Run-time checks can detect errors that cannot be detected at compile-time, including array bound violations, invalid pointer accesses, and use of uninitialized variables. Extensive run-time checking provided by diagnostic compilers incurs significant run-time costs. In [Ste92], 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* [HJ92].

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 [MLS90] 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.

The goal of this work is to provide run-time checking at a very low overhead so that heavily-used programs can be run with checking enabled all the time. Run-time checking does not involve modification of program variables and it does not change the result of valid computations. Hence

*© 1994 by Harish Patil and Charles Fischer.
This work was supported by NSF grant CCR-9122267

run-time checking can be overlapped with normal computation. One approach to concurrent run-time checking is to use specialized hardware. Tagged hardware [Feu73] can be used for type-checking at run-time. Watchdog processors [MJ88] are used to provide control flow checking. 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). We propose to use general purpose multi-processors for concurrent run-time checking.

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 be able to incorporate as many as four general-purpose central processing units on a single chip [GGPY89]. We believe it is quite likely that current applications will not be able to *routinely* use this extra processing power. As a result, processors will often be underutilized. We plan to use spare processors to execute run-time checks using a technique called *shadow processing*.

The basic idea in shadow processing is to partition an executable program into two run-time processes, derived from the same source program. One is the **main process**, executing as usual, without run-time checking. The other is a **shadow process**, following the main process and performing the run-time checks needed to validate its performance. Figure 1 shows a generic shadow

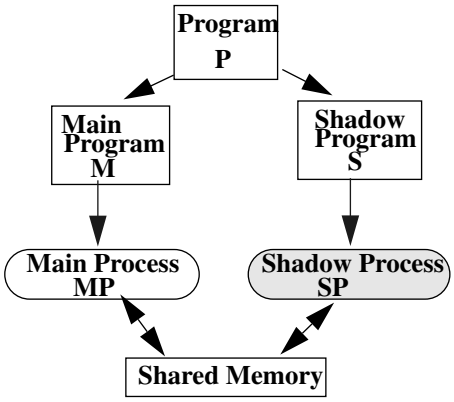

FIGURE 1. Shadow Processing

processing environment.

One key issue in shadow processing 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 (say 5-10%) to justify the use of shadow processing for heavily-used programs.

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.). The

shadow process 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. 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 and those that affect flow of control. Hence a shadow process need not greatly lag behind the main process; it may be able to detect errors in almost real time.

We have developed a prototype shadow processing system for checking pointer and array accesses in C programs. We call our checking technique *shadow guarding*. The highlights of this work are:

1. **Low overhead to the original program:** The main process which computes results runs almost as fast as the original process. Run-time checking is taken out of the critical execution path of the main process. The overhead to the main process (which the user sees) is less than 10% as opposed to typical overhead of 300-500% with traditional approaches. Thus there is no need to turn off run-time checking.
2. **Fast error reporting:** The shadow process needs to perform only the computations relevant to run-time checking. Thus a shadow process co-operating with a main process can terminate much earlier than a single process performing both the computation and checking. Sometimes shadow can even run ahead of the main process catching errors before they actually occur.
3. **Handling of real programs:** Shadow guarding has been implemented to handle "real" C programs. It provides *complete* [ABS94] error coverage for array and pointer access errors. It has been used to detect previously unreported errors in a number of SPEC benchmarks and Unix utilities.
4. **Many applications:** A shadow program is an abstract version of the main program, executing only those statements that are relevant for the activity being monitored. Shadow processing seems well suited for any run-time analysis of programs that can be carried out on abstract versions of the programs. Typical applications include 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 rest of this report is organized as follows. Section 2 presents details of shadow guarding. Section 3 discusses the performance of our prototype and presents techniques to reduce the shadow processing overhead. Section 4 presents related work. Finally, Section 5 summarizes our results.

2 Shadow Guarding

The process of checking the validity of pointer and array accesses using shadow processing will be called *shadow guarding*. This involves creating shadow objects (*guards*), to maintain bounds for pointers and arrays in the main process. Common pointer operations such as dereference and assignment in the main process have shadow counterparts involving *guards*.The key idea in shadow guarding is that in order to check the validity of a pointer dereference, the actual numeric value of the pointer is unimportant. One needs to merely check that the pointer points within its intended referent. Thus the pointer values from the main process need not be examined by the

shadow. This drastically reduces communication between the two processes.

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

2.1 Guards

Every pointer `p` in the main process has a guard `G_p` in the shadow. We will use the term pointer to denote array references as well because when an array identifier appears in an expression, the type of the identifier is converted from "array of T" to "pointer to T"[HS91]. Operations on pointers in the main process lead to operations on guards in the shadow. Structures and unions containing pointers have shadow objects containing guards. Each level of a multilevel pointer type in the main program leads to a `typedef` of corresponding guard type. For example the type "`char ***`" leads to a `typedef` of three guard type "`a_char_guard`", "`b_char_guard`" and "`c_char_guard`" corresponding to "`char *`", "`char **`", and "`char ***`".

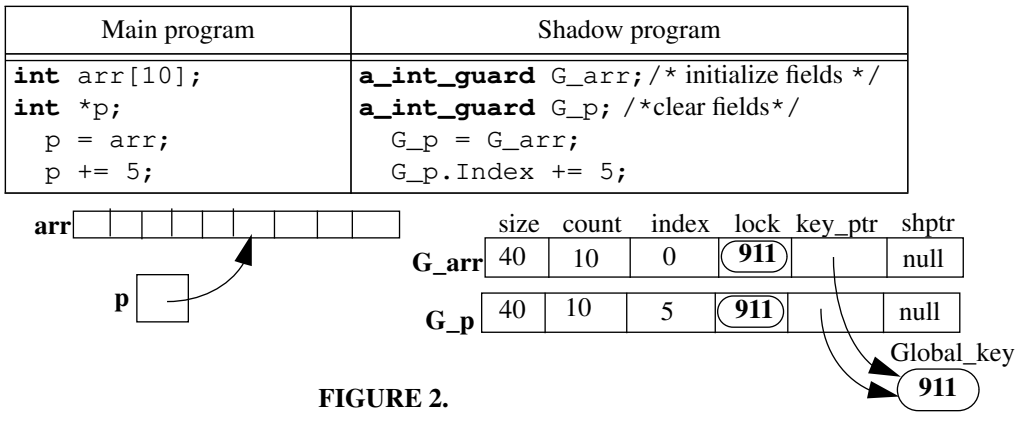

FIGURE 2.

Given the declaration **`T`** `*ptr;` the guard `G_ptr` in the shadow has the following fields:

`Size:` The size, in bytes, of the intended referent of `ptr`. In Figure 2, `G_arr.Size` is set to 40 because `arr` has 10 elements of size 4 (`sizeof(int)`) each.

`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 shadow due to the statement "`p +=5`" in the main.

Lock: An identifying code used to check temporal legality of dereference of `ptr`.

Key_Ptr: This field points to the identifying code of an object.This code must match 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 and 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`).

Main program	Shadow program
`char **p,*q, c;`	`b_char_guard G_p;`
	`a_char_guard G_q;`
`q = &c;`	`/*set fields of G_q*/G_q.shptr = NULL;`
`p = &q;`	`/*set fields of G_p*/G_p.shptr = &G_q;`

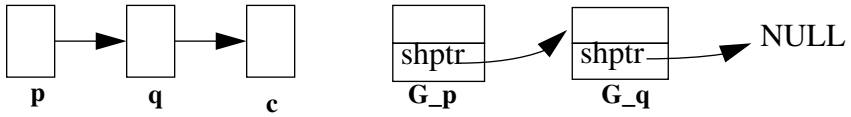

FIGURE 3.

In C, *invalid pointers* [HS91] can be created by casting arbitrary integer values to pointer types, by deallocating 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] in the main program leads to two run-time checks in the shadow 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`. The first check is equivalent to "`0 <=(G_p.Index+i) < G_p.Count`".

2.2 Shadow heap

To be able to check accesses to heap objects in the main process, a data structure called *shadow heap* is maintained in the shadow process. The shadow heap is an expandable array of unsigned integers. Each heap object in the main 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 main, 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 dereference `(*p)` of a pointer whose referent has been freed will go undetected. We shall consider the problem so small that it may be safely ignored.

Pointer operation	Guard operation

`p = malloc(S)`	`G_p.Size = 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.Size`, and `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 reported by requiring that `G_p.Key_ptr` points into the shadow heap.

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, a 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

---

* *Prob(undetected invalid dereference) =*
*Prob(invalid deref) * Prob(two objects getting the same slot) * Prob(two objects getting the same key)*
This will happen only when the slot occupied by *p*'s defunct referent gets reused by another object which happens to get the exact same key value – due to the wrap around of *HeapKey.*

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 main process, `G_p.Key_ptr` is made to point to the shadow stack slot corresponding to `var`'s enclosing frame (global variables use a special `Global_Key`). 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.

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 nonlocal jumps [HS91]. **`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 a calling **`setjmp`** and restores the environment stored in that buffer. Since many active frames on the stack may become inactive after a **`longjmp`**, corresponding slots in the shadow stack are popped and erased (as a result of doing **`longjmp`** in the shadow)

2.4 Implementation

An overview of our prototype shadow processing system is shown in Figure 4. The frontend for shadow guarding reads in a C program and creates the shadow and the main programs. The main program uses a shared circular buffer to communicate values which can not be recomputed in the shadow.

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 shadow 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[HS92]. 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

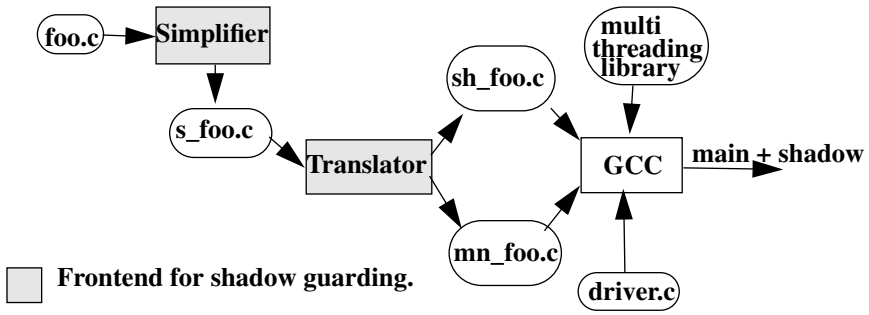

FIGURE 4. Overview of shadow guarding system

number of temporary variables are introduced. These variables increase the demand on register allocation. Hence a simplified program (compiled with `-O4` using `gcc 2.5.8`) runs around 1-2% slower than the input program on a SPARC 630 MP.

The translator in Figure 4 reads in a simplified program and produces main and shadow programs. The main program is obtained by instrumenting the simplified program to communicate selected relevant values. The shadow program is a copy of the simplified program which reads the same selected values from a shared buffer. In addition it performs computations necessary for guarding. Currently, we perform checks only for user defined functions for which source code is available and we provide a clean interface for the library functions. Memory allocation routines such as **`malloc`** and **`free`** are treated separately. Many library functions, such as **`atoi`** and **`strlen`** that do not affect any data structures inside the library are safely replicated inside the shadow process. Special action is necessary for functions such as **`strtok`** that affect internal data structures of the library. Routines that cause output or change the external environment in some way and the routines which may lead to system calls are not repeated in the shadow. Return values of such routines are communicated. Thus execution of the shadow does not ever affect the output of the main process.

3 Results

Shadow guarding was implemented on a Sun SPARC 630 MP which is a 4 SPARC processor shared memory machine running SunOs Release 5.3 [UNIX system V release 4]. Sun's multi-threading library [PKB+91] was used to create main and shadow threads. The test programs included integer benchmarks from the SPEC92 test suite, one program, *yacr*, from the SafeC [ABS94] test suite and many commonly used utilities from SunOs 4.1.3. A utility called *fuzz* [MLS90] was used to generate random input for the SunOs utilities. SPEC benchmarks were tested with their reference inputs.

3.1 Errors uncovered

Run-time errors which do not crash programs can go unnoticed for a long time. Shadow guarding reports such errors as they occur. Programmers can use the feedback from shadow guarding to eliminate subtle bugs. We uncovered unreported errors in five test programs which did not crash.

These programs (with the number of errors in parentheses) were *decompress*(1) and *sc*(2) from SPEC92 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 [MLS90] in earlier versions of SunOs. However, we found new errors in these utilities as well. In all, we uncovered 15 errors in nine programs. A complete report on the errors in the programs tested so far is presented in the Appendix. We expect further testing will certainly reveal errors in other widely-used programs.

3.2 Performance

Table 1 compares (real) execution time of main process with that of the original program. The overhead to the main process is mainly due to the communication with the shadow process. The overhead indicates the delay in obtaining results of the original computation in the shadow processing environment. It is below 10% in all the cases. To the average user, the main process will appear almost indistinguishable from the original uninstrumented program.

Program	espresso	alvinn	yacr	eqntott	xlisp	compress	sc	decompress
original	6s	265s	35s	67s	12s	13s	7s	17s
main process	6s	268s	35s	68s	13s	14s	7s	17s
Overhead	**0%**	**1.13%**	**0%**	**1.49%**	**8.3%**	**7.7%**	**0%**	**0%**

Table 1: Shadow guarding: overhead to the user program

How fast does the shadow run? We found that for *decompress*, the shadow actually ran faster than the original program. In other cases, the shadow ran 1.2 to 8.5 times slower than the original program. These slowdowns are still less than the slowdowns (1.5 to 10.1) of the input program instrumented to perform the checks sequentially. Figure 5 compares execution time of shadow guarding with that of sequential checking. Since the shadow process executes in the background, not interfering with the user, it may complete *after* the user has gone on to another task. The user sees only the main process, which is essentially as fast as the original program.

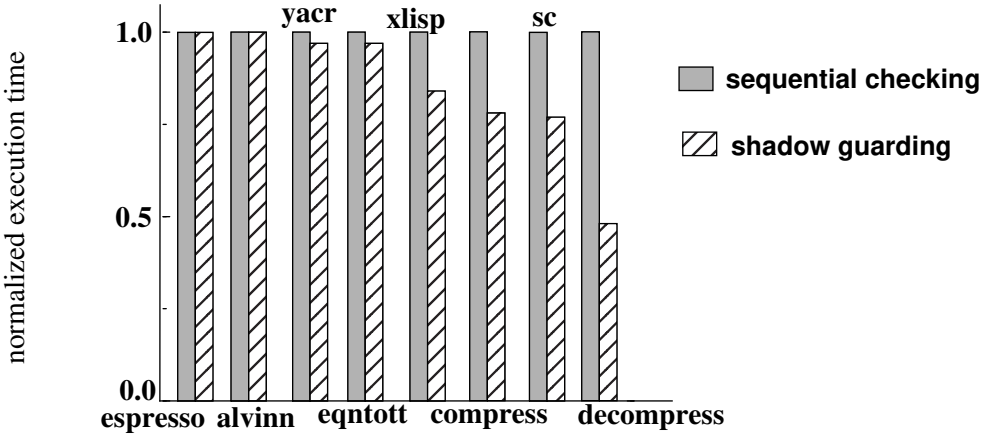

FIGURE 5. Execution time: shadow guarding vs. sequential checking

3.3 Speeding up shadowing

The execution time of the shadow process depends on two factors:

Number of pointer operations in the input program. To reduce this slowdown, a variety of range checking optimizations such as [Gup93] can be applied to reduce the number of checks done at run-time. Both the sequential approach and shadow guarding approach can equally benefit from these optimizations.

We are currently exploring an optimization to reduce the overhead of temporal checks. We want to statically determine the life-time of the referent of pointers. The approach based on [EGH94] and [And93] classifies pointers into the following categories:

i. Points_to_NULL: These are uninitialized pointers whose dereference is reported at compile time.

ii. Points_to_global

iii. Points_to_local_of_current_function

iv. Points_to_local_of_exited_function: Dereferencing these pointers is illegal and can be reported at compile time.

v. Points_to_heap.

Temporal checks and maintenance of temporal attributes can be elided for the pointers in the second and third category – sometimes cutting the runtime overhead by half.

Extent of repetition of computations from the input program. The shadow guarding process needs to recompute only those values that affect the flow of control and pointer values being monitored. Any computation that affects the external environment (e.g. **`printf`**, **`chdir`**) may not be repeated. The effect of deleting such computations is very apparent for *compress*, *sc* and *decompress* in Figure 5. The shadow took less time to execute than the sequential approach mainly because it could avoid all the output calls. The effect was most pronounced for *decompress* where the shadow not only beat the sequential instrumented approach but it was also faster than the original program.

There are cases in Figure 5 in which the shadow takes almost as much time as the sequential approach. These programs do not spend much time in the output routines hence the effect of reducing output calls is not apparent. We think the speed of shadow can be further improved by using slicing technology to remove computations not necessary for guarding. We modified the shadow program for *alvinn* to delete statements not necessary for guarding and the resulting shadow process ran 21% faster than the sequential checking process. We are currently interfacing our translator with the slicing backend from the Wisconsin Program-Integration System. The slicing backend, based on [BH93], can slice programs with arbitrary control flow. It should allow us to generate much faster shadow programs.

4 Related work

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 [SM93]. Concurrent runtime 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.

CodeCenter[KLP88] 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 [HJ92] is a commercially available system that modifies object files to, in essence, implement a byte-level tagged architecture in software. It maintains a bit table at run-time to hold a two-bit state code for each byte in the memory. A byte could be in one of the three states i) unallocated, ii) allocated but uninitialized and iii) allocated and 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.

Austin *et al* [ABS94] have proposed translation of C programs to *SafeC* programs 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. Shadow guarding shares the "completeness" of error detection with *SafeC*. Unlike the system described in [ABS94], shadow guarding is completely automated and has been used to detect errors in "real" programs. 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 capa-

bility database. Such a search is avoided in shadow guarding by adding the `Key_ptr` field in guards. The value of `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.

Shadow processing was motivated, in part, by a tool called AE that supports *abstract execution* [Lar90]. AE is used for efficient generation of detailed program traces. A source program, in C, is instrumented to record a small set of key events during execution. After execution these events serve as input to an abstract version of the original program that can recreate a full trace of the original program. The events recorded by the original program include control flow decisions. These are essentially the same data needed by a shadow process to follow a main process. AE is a *post-run* technique that shifts some of the costs involved in tracing certain incidents during a program's execution to the program that uses those incidents. In contrast shadow processing is a *run-time* technique that removes expensive tracing from the critical execution path of a program and shifts the cost to another processor.

Parasight [AG88] 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 already executing main process when a processor becomes available, "spot checking" the main program. The shadow will have to start executing at certain well defined points, say at the entry of functions. The main process will have to leave a trail of indicators of having reached those points (in a shared buffer).

5 Conclusions

With shared memory multiprocessors becoming increasingly common, run-time checking techniques that exploit multiple processors become attractive. Shadow guarding is a technique that uses shared memory multiprocessors to check the validity of array and pointer accesses. Current approaches to pointer access checking work sequentially, typically slowing a computation 3-4 times. Such high overheads make those approaches unsuitable for use with 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 elaborate, sometimes delayed, error report 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 which do not crash the original program can be extremely helpful in correcting the program.

We have developed a prototype shadow guarding system which supports full-size programs written in C. Our system instruments an executable user program in C 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 and performs run-time 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 input program, it runs up to two times faster than a single process performing both the computation and checking. Sometimes the shadow process can even run ahead of the main process catching errors before they actually occur. Our system has found a number of errors (15 so far) in widely-used Unix utilities and SPEC92 benchmarks. We believe our approach shows great potential in improving the quality and reliability of application programs at a very modest cost.

Acknowledgments

We thank Bart Miller for making the machine *shemesh* available for this study and Krishna Kunchithapadam for providing OS support. We also thank Tom Reps and Susan Horwitz for allowing access to WPIS and Rosay Genevieve for her help in setting up WPIS. Steven Kurlander, Manuvir Das, Madhusudhan Talluri, Krishna Kunchithapadam, Brad Richards, Guhan Viswanathan, T. N. Vijaykumar, and Satish Chandra provided useful comments on an earlier draft of this report.

References

[ABS94] 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.

[AG88] Ziya Aral and Ilya Gertner. High-level debugging in Parasight. In *ACM Workshop on Parallel and Distributed Debugging*, pages 151–162, May 1988.

[And93] Lars Ole Andersen. Binding-time analysis and the taming of C pointers. In *Proceedings of the ACM SIGPLAN'93 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 47–58, 1993.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, volume 749. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993.

[EGH94] Marayam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and*

Implementation, pages 242–256, 1994.

[Feu73] E. A. Feustal. On the advantages of tagged architectures. *IEEE Transactions on Computers*, C-22(7):1241–1258, July 1973.

[GGPY89] Patrick Gelsinger, Paolo Gargini, Gerhard Parker, and Albert Y. C. Yu. Microprocessors circa 2000. *IEEE Spectrum*, pages 43–47, October 1989.

[Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2:135–150, March-December 1993.

[HJ92] 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.

[HS91] Samual P. Harbison and Guy L. Steele Jr. *C - A Reference Manual*. Prentice Hall, 3rd edition, 1991.

[HS92] Laurie J. Hendren and Bhama Sridharan. The SIMPLE AST - McCAT compiler. ACAPS design note 36, School of Computer Science, McGill University, Montreal, Canada, 1992.

[KLP88] 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.

[Lar90] James Larus. Abstract execution: A technique for efficiently tracing programs. *Software - Practice and Experience*, 20(12):1241–1258, December 1990.

[MJ88] 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.

[MLS90] 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.

[PKB+91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter USENIX Conference*, pages 1–14, 1991.

[SM93] Sriram Sankar and Manas Mandal. Concurrent runtime monitoring of formally specified programs. *Computer*, 26(3):32–41, March 1993.

[Ste92] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software - Practice and Experience*, 22(4):825–834, April 1992.

Appendix
Bug Report from Shadow Guarding

1 Setup

Shadow guarding was implemented on a SPARC 630 MP which is a 4 SPARC processor shared memory machine running SunOs Release 5.3 [UNIX system V release 4]. Sun's multi-threading library[PKB+91] was used to create main and shadow threads. The test programs included integer benchmarks from the SPEC92 test suite, one program from the SafeC test suite and a number of commonly used utilities from SunOs 4.1.3. A utility called *fuzz* [MLS90] was used to generate random input for the SunOs utilities. SPEC benchmarks were tested with their reference inputs.

Utility	Source	# of different errors detected	Program crashes?
cb	SunOs 4.1.3	1	no
col	SunOs 4.1.3	1	yes
decompress	SPEC92	1	no
deroff	SunOs 4.1.3	3	yes
ptx	SunOs 4.1.3	4	no
sc	SPEC92	2	no
ul	SunOs 4.1.3	1	no
uniq	SunOs 4.1.3	1	yes
units	SunOs 4.1.3	1	yes

Run-time errors which do not crash programs can go unnoticed for a long time. Shadow guarding reports such errors promptly after they occur. Programmers can use the feedback from shadow guarding to eliminate subtle bugs. We uncovered unreported errors in five utilities (which did not crash). The SunOs utilities which crashed were already reported to be buggy [MLS90] in earlier versions of SunOs. However, we found new errors in these utilities as well. A complete report on the errors in the utilities tested so far is presented below. We expect further testing will certainly reveal errors in other utilities.

2 Detailed error report

cb

Macros `isop, isalpha, isupper ...` all index the array `_chtype` using character variable '`c`'. The size of the array is 129, with the assumption that input characters will be in the range 1-128. With random input, '`c`' may not be ASCII, violating bounds of array '`_chtype`'.

col

```
229     c3 = *line;
```

Inside the program, there is a lot of places where the pointer 'line' is incremented without checking its validity.

decompress

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 #of_bits/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.

Check: Put 'assert((bp-&buf[0])<BITS);' before line 1145.

derof

```
341   *++lp = C;
345     *lp = '\0';
```

The character pointer 'lp' is incremented without being checked. Whenever the line being processed is longer than 512 characters, there will be a pointer out of bound error.

```
812   .. chars[cp[0]]==LETTER
```

Array 'chars' is indexed using input characters assuming that they are ASCII. For random input many non-ASCII characters result in violation of array bounds for 'chars'.

ptx

All the errors detected are due to accesses beyond bounds of global arrays. The reason they did not crash the program must be that the errant accesses (illegally) referred to other valid global data.

This program has "#define isbreak(c) (btable[c])"
where char btable[128] is used to indicate whether a character in the range 0-127 is a break character or not. The assumption here is that input characters will all be printable characters in the range 0-127. If the utility is fed a random stream of characters, the check isbreak(c) violates the bounds of array btable.

```
332 if(isabreak(*pchar)) {
338 if(isabreak(*pchar++))
```

Function getline() uses the pointer linep to run through the array
"char line[200]". linep is incremented and dereferenced many times in the function with-

out checking whether input line is small enough to fit in 200 characters.

```
308 *++linep = '\n';
```

Similar error occurs in function `getsort()`

```
469  *linep++ = c;
```

Function `getsort()` has the following code:

```
407       linep = line;
408       while((c = getc(sortptr)) != EOF) {
409           switch(c) {
410
.
415           case '\n':
416               while(isspace(linep[-1]))
```

If `c` is `'\n'` on the very first iteration, `line[-1]` gets accessed.

sc

Found error is `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`).

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.

ul

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' gets larger than the maximum index (511) allowed for 'obuf'.

units

```
199       *cp++ = c;
```

Pointer "cp" is not checked for before it is accessed producing out of bounds pointer reference

uniq

```
76        *buf++ = c;
```

Size of the buffer 'buf' is not checked before this statement, for input line size greater than 1000, array bound is violated.

References

[MLB90] Barton P. Miller, Fredriksen Lars, and So Brian. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[PKB+91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter USENIX Conference*, pages 1–14, 1991.