Presented at AADEBUG'95

Efficient Run-time Monitoring Using Shadow Processing

Harish Patil* and Charles Fischer
University of Wisconsin —Madison**

Abstract

General purpose multiprocessors are becoming increasingly common. We propose using pairs of processors, one running an ordinary application program 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 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, checking pointer and array accesses and detecting memory leaks. 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. Our system has found a number of errors (15 so far) in widely-used Unix utilities and SPEC92 benchmarks. It also detected many subtle memory leaks in various test cases. We believe our approach shows great potential in improving the quality and reliability of application programs at a very modest cost.

1 Introduction

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 [GGPY89].

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. Our department has more than 60 dual processor workstations. We monitored 18 of those machines over a period of 24 hours using 'condor'[LLM88]. 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 a technique called *shadow processing*.

The basic idea is to partition an executable program into two run-time processes as shown in Figure 1. One is the **main process**, executing as usual. The other is a **shadow process**, following the main process and performing auxiliary tasks such as run-time checking and profiling. The two processes may communicate and synchronize during execution. The model of shadow processing in Figure 1 is very general; the input program may be in any language (including machine 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 which 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

*This work was supported by NSF grant CCR-9122267.
**Authors' address: Computer Sciences Dept., 1210 W. Dayton St., Madison, WI 53706, USA.
Email: *{patil,fischer}@cs.wisc.edu*

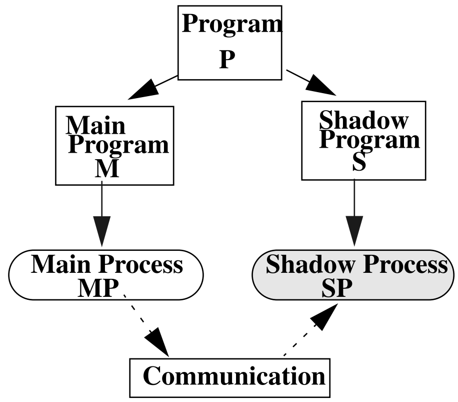

FIGURE 1. Shadow Processing

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 which 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 which 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 which affect the analysis being performed. This can be achieved by taking a program slice[Tip94] with the slicing criteria involving the variables required for the analysis. These slices can be combined to form the shadow program.

1.1 Shadow run-time checking

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 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.

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.

1.2 Shadow Guarding: Overview

We have developed a prototype shadow processing system for run-time checking of array bound violations and invalid pointer accesses in C programs. We call our checking technique *shadow guarding*. It 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*.

Our prototype shadow guarding system is a source-to-source translator for C programs. An input C program is used to generate a main program and a shadow program. Functions and statements in the main program have shadow counterparts. In addition, the shadow program maintains attributes for pointers in the main program using guards. These attributes are used to check legality of pointer dereferences in the main program. We believe the overhead to the main process must be very modest (say 5-10%) to justify the use of shadow guarding for heavily-used programs. Hence a shadow process generated by our prototype computes all the values it can, minimizing the burden placed on the main process. Special handling is required for calls to external functions not defined by the user. Calls such as **printf** which affect the external environment are skipped in the shadow. Return values of some external functions may be needed to determine the control flow of the shadow process. Such values are communicated via a shared circular buffer.

The highlights of this work are:

1. **Low overhead to the original program:** The main process that computes results runs almost as fast as the original process. Memory access 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%. Thus there is no need to turn off run-time memory access checking.
2. **Fast error reporting:** The shadow process needs to perform only the computations relevant to memory access checking. Thus a shadow process co-operating with a main process can run much 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.
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.

The rest of this report is organized as follows. Section 2 presents details of shadow guarding. Extensions to shadow guarding for memory leak detection are presented in Section 3. Section 4 presents a few details of our prototype. Section 5 discusses the performance of our prototype and presents techniques to reduce shadow processing overhead. Section 6 presents related work. Finally, Section 7 summarizes our results.

2 Guards

Each pointer `p` in the main process has a guard `G_p` in the shadow. The guard for a pointer stores spatial and temporal attributes for the pointer's referent. The idea is similar to safe pointers used in [ABS94]. We will use the term pointer to include array references as well because when an array identifier appears in an

Main program	Shadow program
`int arr[10];`	`a_int_guard G_arr;/* initialize fields */`
`int *p;`	`a_int_guard G_p; /*clear fields*/`
`p = arr;`	`G_p = G_arr;`
`p += 5;`	`G_p.index += 5;`

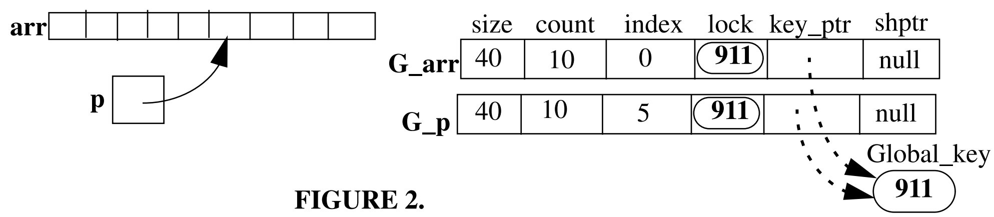

FIGURE 2.

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.

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.
   The attributes necessary to verify that a pointer dereference is spatially valid include the size of 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 main process. These keys are stored in some auxiliary data structures in the shadow process. These 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 pointer to the location where the key is stored.

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

`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 2.1 and 2.2.

`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; /* 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;`

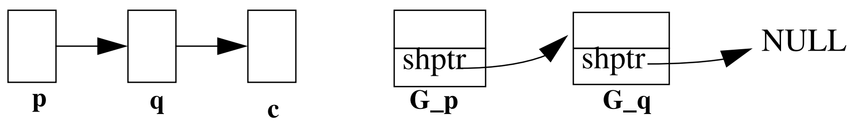

FIGURE 3.

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`".

2.1 Shadow heap

To be able to check accesses to heap objects in the main process, a data structure called the *shadow heap* is maintained in the shadow process. The shadow heap is an expandable array of unsigned integers. Each heap object in the main process 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 process, 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 chance so small that it may be safely ignored. Actions on a dynamic allocation are explained below:

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 detected by requiring that G_p.key_ptr points into the shadow heap.

2.2 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 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.

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 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. 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 shadow.

3 Detecting Memory Leaks

A memory leak is a block of memory that is allocated but is no longer accessible (and hence lost). Memory leaks in long-running programs add up and may ultimately exhaust the available memory. Garbage collection [Wil] is a well-known technique for preventing memory leaks by reclaiming unusable memory blocks. Garbage collectors are routinely used for languages such as ML and LISP. Garbage collection in C is complicated by features such as pointer arithmetic, casting, and unions [Det90].

We have modified our shadow guarding system to detect memory leaks in C programs. We use the shadow process to perform concurrent garbage detection. Our approach is based on reference counting garbage collection[Wil]. The shadow process maintains a count of references to each heap object allocated in the main process. These reference counts are maintained in a data structure very similar to the shadow heap. A new field refcount_ptr of the type "unsigned int *" is added to shadow guards for pointers in the main process. This field is initialized to `NULL`. For a pointer `p` in the main process referring to a heap object *O*, `G_p.refcount_ptr` points to the reference count for *O*.

An assignment to a pointer `p` in the main program leads to a garbage detection action involving the corresponding guard `G_p` in the shadow program as shown in the table below. Before a function returns, reference counts of referents of all the local pointers of the function must be decremented. Similar action is necessary for local pointers in the scopes exited using a `break`, `goto`, or `longjmp()`

Pointer operation	Guard operation

`p = <expr>`	/* Check if the assignment creates garbage */
	if(G_p.refcount_ptr!=NULL){
	  /*p points to a heap object */
	  *(G_p.refcount_ptr) -=1;
	    if(*(G_p.refcount_ptr)==0){
	        GARBAGE_REPORT_ACTION
	    }
	}
	
	/* Perform the assignment to G_p from the attributes of <expr> */
	if(G_p.refcount_ptr!=NULL){
	  /*After assignment p points to a heap object*/
	    *(G_p.refcount_ptr) +=1;
	}

p = **malloc**(S)	/*Check if the assignment creates garbage*/ /* (as above) */ /* set other fields of G_p */ G_p.refcount_ptr = **Next_refcount_slot**(); *(G_p.refcount_ptr)= 1;

Freeing an object eliminates one way of accessing pointers embedded in it. Hence after a call `free(p)` the reference counts for all the heap objects being pointed to from within the referent of `p` are decremented as shown below.

Pointer operation	Guard operation

free(p)	if(G_p.refcount_ptr==NULL){
	   report("p's referent can not be freed");
	}else{
	  /*p points to a heap object */
	      if(*(G_p.refcount_ptr)>1){
	         report("free leaves dangling references");
	     }
	}
	/* Using G_p.shptr recursively decrement refcounts of referents of embedded pointers */
	
	/* if p is a multilevel pointer or a pointer to a struct, G_p.shptr needs to be freed as well */
	if(G_p.shptr) **free**(G_p.shptr);
	
	/*Reset fields of G_p*/

A well known problem with reference counting garbage collection is its inability to reclaim circular data structures which have become garbage[Wil]. Any attempt to resolve this problem either by extending reference counting [Chr84] or by moving to a "mark and sweep" or a "copying" garbage collection scheme[Wil] will require cooperation from the main process. We currently ignore memory leaks due to circular data structures.

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

```
union u{
   struct s1 *field1;
   struct s2 *filed2;
   int *field3;
}*pu;
```

Suppose pu is a pointer to the union described above. If the object pointed to by pu becomes garbage the recursive garbage detection inside the referent of pu depends on the type of the current active field of the union. We maintain a run-time tag indicating the type of the currently active field of a union in the shadow object for the union. In the example above, a garbage report for pu will lead to an appropriate recursive garbage detection involving the guard for its active pointer field. The run-time type tag is also useful in detecting the "misuse" [HS91] of the value of one field of a union as another field.

4 Implementation

An overview of our prototype shadow processing system is shown in Figure 4.

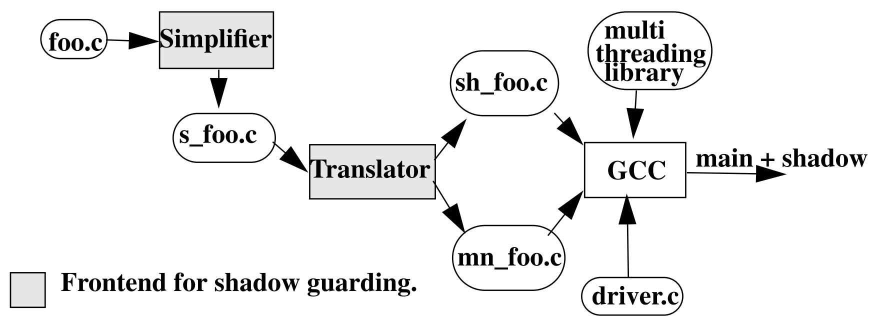

FIGURE 4. Overview of shadow guarding system

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 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 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 communicates certain relevant values (to be described in Section 4.1) using a shared circular buffer. The shadow program reads the relevant values and performs computations necessary for guarding.

4.1 Handling external functions

Currently, we perform checks only for user defined functions for which source code is available and we provide a clean interface for external functions. 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). 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.

We classify calls to external functions in three categories:

Output calls: These are calls to library functions such as **`printf`** or system calls such as **`chmod`** which affect the external environment. Clearly, these calls should not be repeated in the shadow program because the execution of the shadow should never affect the output of the main process.

Safe calls: These include library routines such as **`atoi`** which do not lead to system calls hence can be safely replicated in the shadow program if needed.

Unsafe calls: These are system calls and library calls which may result in a system call. The shadow process reads the return values of such calls from the main process if needed.

We have a database of categories of external function calls which we keep updating as new external function calls are encountered.The translator in Figure 4 uses this database while generating main and shadow programs. The main program is obtained by instrumenting the simplified program to communicate return values of unsafe calls. The shadow program is a copy of the simplified program without the output calls. It reads the return values of unsafe calls from a circular buffer shared with the main program. In addition it performs computations necessary for guarding.

Our prototype requires that the input program does not cast non-pointer values into pointers. Casting a low level pointer to a higher level is also prohibited. We had to modify some of the test programs to handle such cases. 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.

Currently, we support only single process programs i.e. programs calling **`fork`**, **`exec`** are not supported. We can envision creating a new shadow process for each child of the main process. Handling signals and signal handlers is hard. We have to make sure that every signal received by the main process is delivered to the shadow process as well so that the shadow can perform memory access checking in user defined signal handlers. We currently ignore this problem. The library function **`qsort`** takes in a pointer to a comparison function provided by the user. Any calls to this user function from **`qsort`** will not be shadowed since the call to **`qsort`** itself is not shadowed. Our translator disables any communication in every comparison function used in **`qsort`**.

5 Results

Shadow guarding and memory leak detection were implemented on a Sun SPARC 630 MP which is a 4 SPARC processor shared memory machine. It has 56Mbyte of real memory and it runs SunOS Release 5.3 [UNIX system V release 4]. Sun's multi-threading library 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.

5.1 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 can be found in [PF94]. We expect further testing will reveal errors in other widely-used programs.

5.2 Overhead to the user program

Table 1 compares (real) execution time of the main process with that of the original program. The overhead to the main process is due to communication and any memory conflicts 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

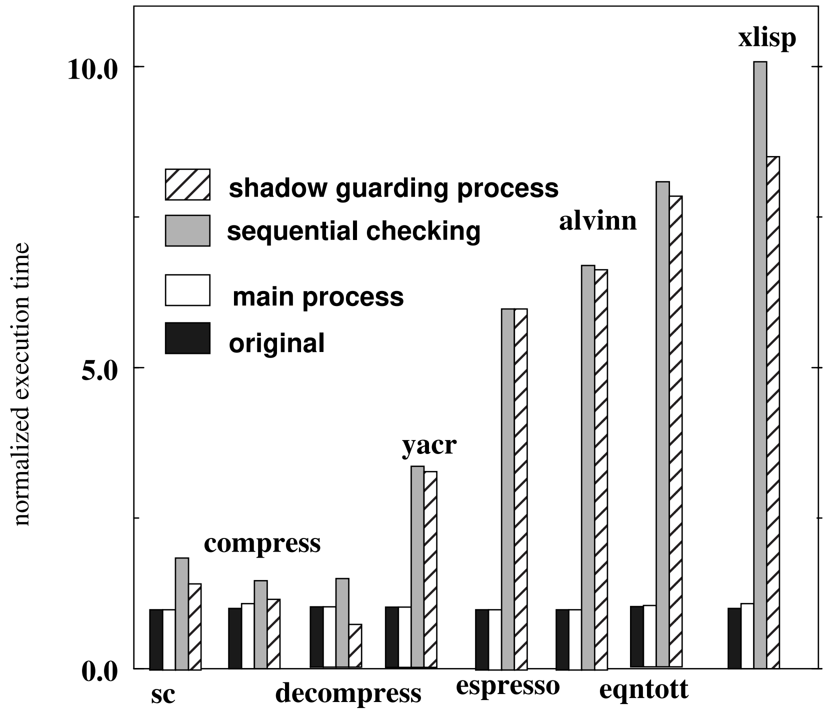

FIGURE 5. Performance of shadow guarding

indistinguishable from the original uninstrumented program.

Program	espresso	alvinn	yacr	eqntott	xlisp (8-queen)	compress	sc	decompress
original	6s	265s	35s	67s	12s	13s	7s	17s
main process	6.5s	268s	37s	69s	13s	14s	7s	18s
Overhead	**8.3%**	**1.13%**	**5.7%**	**2.98%**	**8.3%**	**7.7%**	**0%**	**5.88%**

Table 1: Shadow guarding: overhead to the user program

A circular buffer of size 64K was used to communicate return values of unsafe external function calls. This buffer size was sufficient for all the test cases except *compress*. The main process for *compress* reads its input using **`getchar`**. We treat **`getchar`** as an unsafe external call whose return value is explicitly communicated to the shadow process.The reference input file for compress has one million bytes leading to a large number of values communicated between the main and the shadow. But this was not a problem since the shadow was able to consume old values from the circular buffer before the main process overwrote them.

In general we have found that input routines such as **`getchar`** are very often the major source of communication. If the input is from a file, the shadow program can read its own input through a link to the file (we took this approach for testing *compress* and *decompress*, eliminating almost all the overhead to the main process). If the main process modifies its input file a separate copy instead of a link will be needed for the shadow. Since the shadow program never writes into a file, it will never affect the output of the main process. The main process however has to communicate input arriving from standard input to the shadow explicitly. In some cases the shadow process may not be fast enough to keep up with the main process. We could make the main process busy-wait for the shadow in such cases slowing down the main process. Alternately, we can create another "hungry" shadow process whose job is just to read values from the shared buffer and to stash them away for the shadow process performing memory access checking or any other analysis.

5.3 Performance

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 summarizes the performance of shadow guarding. The user sees only the main process, which is essentially as fast as the original program. The shadow process executes in the background, not interfering with the user, though it may complete *after* the user has gone on to another task. However, all the errors we discovered were reported by the shadow before the main process had finished.

Our frontend for shadow guarding has been functional for 8 months. Shadow guarding has been successful in detecting bugs in many real programs. A technical report describing shadow guarding is now available[PF94]. We have since then modified our prototype to detect memory leaks as outlined in Section 3. Table 2 presents the cost of memory leak detection. Test cases *compress*, *decompress*, and *alvinn* do not need memory leak detection because they do not allocate memory dynamically. Memory leak detection requires much more work than shadow guarding hence it is more expensive. However its cost is hidden from the user since memory leak detection is performed using a shadow process.

Table 2: Guarding + Memory leak detection: execution time in seconds

Test case	Original program	Main program	Shadow program	Sequential memory leak detection
yacr (input2)	35	37	183	184
espresso (opa.in)	6	6.5	55	58
eqntott (int_pri_3)	67	69	1115	1126
xlisp (6 queens)	3	3	58	61
sc (load1.in)	7	7	24	26

5.4 Speeding up shadowing

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

Number of pointer operations in the input program. To reduce slowdown due to pointer checking, 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 the 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] 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.

Repetition of computations from the input program. The shadow guarding process needs to recompute only those values that affect the pointer values being monitored. Any computation that affects the external environment (e.g. `printf`, `chmod`) must 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 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. The resulting shadow process ran 21% faster than the sequential checking process. We are currently interfacing our translator with a slicing backend that is an offshoot of the Wisconsin Program-Integration System [Rep89]. Slicing should help shadow guarding in two ways. First since the shadow may not need all the input values and return values of all the unsafe calls, the number of communication calls in the main will be reduced. Second, deleting irrelevant computations from the shadow program will make the shadow run faster and it may be able to keep up with the main process.

6 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 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.

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. 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.

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 an 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).

7 Conclusions

Shadow guarding is a technique that uses idle processors in multiprocessor machines 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, checking pointer accesses and detecting memory leaks. 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 Genevieve Rosay for her help in setting up WPIS. Thanks to Miron Livny and Jim Pruyne for their help with condor. Susan Horwitz, Jim Larus, and Steve Kurlander provided useful comments on this paper.

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.

[Chr84] Thomas W. Christopher. Reference count garbage collection. *Software - Practice and Experience*, 14(6):503–507, June 1984.

[Det90] David L. Detlefs. Concurrent garbage collection for C++. CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1990.

[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.

[LLM88] 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.

[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.

[PF94] Harish Patil and Charles Fischer. Shadow Guarding: Run-time checking you can afford. TR 1254, Computer Sciences department, University of Wisconsin, Madison, Wisconsin, November 1994. Available by anonymous ftp from *ftp.cs.wisc.edu*.

[Rep89] Thomas Reps. Demonstration of a prototype tool for program integration. TR 819, Computer Sciences department, University of Wisconsin, Madison, Wisconsin, January 1989.

[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.

[Tip94] Frank Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[Wil] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*. To appear.