A Survey of Systems for Detecting Serial Run-Time Errors

The Iowa State University's High Performance Computing Group
Glenn R. Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Ying Li,
Olga Taborskaia, and Yanmei Wang
{grl, jjc, hoekstra, kraeva, yingli, olga, yanmei}@iastate.edu
Revised February 22, 2006

Abstract: This paper evaluates the ability of a variety of commercial and non-commercial software products to detect serial run-time errors in C and C++ programs, to issue meaningful messages, and to give the line in the source code where the error occurred. The commercial products, Insure++ and Purify, evaluated did the best of all the software products we evaluated. Error messages were usually better and clearer when using Insure++ than when using Purify. Our evaluation shows that the overall capability of detecting run-time errors of the non-commercial products is significantly lower than the quality of both Purify and Insure++. Of all the non-commercial products evaluated, Mpatrol provided the best overall capability to detect run-time errors in C and C++ programs.

I. Introduction

Debugging applications programs can be very time consuming. However, having good software tools can greatly decrease this time. While some program errors can be found at compile time, there are other program errors than cannot be detected until run-time. We call these errors run-time errors. Observe that we assume the language syntax of the program is correct and an executable was created. For example, suppose the value of an integer variable n is not known at compile time. If n is outside the declared bounds of an array A, then an out-of-bounds memory access error will occur during run-time when the program reads A[n]. The value of n may have been read from an input file or the value of n may have been the result of a calculation not performed at compile time.

The purpose of this paper is to evaluate the ability of a variety of software products to detect serial (not parallel) run-time errors, to issue meaningful messages, and to give the line in the source code where the error occurred. The language a program is written in determines the kinds of run-time errors that can occur. We limit our investigation to errors that can occur when executing serial Fortran, C, and C++ programs. However, most of the software currently available for detecting run-time errors only supports C and C++, so we only evaluated selected software products for detecting run-time errors in C and C++ programs. These selected software products were evaluated by writing a variety of C and C++ tests with run-time errors.

Section II gives a list of the commonly-encountered run-time errors that occur for Fortran, C, and C++. Section III lists the software products evaluated. The evaluation was performed in two steps. First information about each software product was obtained from a variety of web sites. However, just because a web site says that a software product detects certain run-time errors, this does not mean it really does detect these run-time errors in a variety of situations nor does it mean that the software provides information that allows the user to easily locate the error. Thus, the second step was to evaluate the ability of these software products to detect run-time errors on a series of C and

C++ run-time error test suites that we wrote. The task of evaluating all software products against our test suites was too large, so only those software products that appeared to be the best were actually evaluated using our tests. These tests are described in section IV.

Sections V and VI contain the results of running these tests using the selected software products. Section VII contains our conclusions.

II. Detecting Run-Time Errors

In this section we list the commonly-encountered run-time errors that occur when executing Fortran, C, and C++ programs. Most compilers and run-time systems provide detection of some run-time errors but only when the program has been compiled with a variety of special options. Having many different compiler options makes it difficult for those trying to debug a program. It would be much easier for users to debug their programs if compilers provided a simple compiler option, e.g. –debug, that turns on all compiler options that are helpful for debugging.

1. Detecting uninitialized Variables

The execution of a program that uses variables before they have been initialized by the program can cause unpredictable results. Using variables prior to their initialization is a common programming error and is often difficult to find without software support to automatically detect this error. Many compilers and run-time systems do provide detection of uninitialized variables.

2. Detecting Overflows, Underflows, and Divide by Zeros

The IEEE standard for floating point operations does not consider overflows, underflows, and divide by zeros to be errors. When an underflow occurs, most people want the variable's value set to zero and execution to continue. This is the default behavior for most and probably all compilers and run-time systems. We do not consider underflows to be an error. However, floating point overflows and divide by zeros usually are considered errors that require changes in the source code. Thus, one would want run-time error detection software to detect floating point overflows and divide by zeros and to show where they occurred.

Some compilers and run-time systems provide detection of floating point overflows and divide by zeros but often not for integer divide by zeros and not for integer overflows.

3. Detecting incorrect argument data types and incorrect number of arguments

Fortran and C functions (and subroutines for Fortran) are not required to have interface blocks/declarators, but C++ does require declarators. When this information is included in a program for the functions used in a program, it allows the compiler (at compile time) to determine if the data types and number of arguments of all function calls in the program are consistent with the information in the interface blocks/declarators. However, the compiler cannot determine if the interface blocks/declarators are consistent with what is contained in the actual function definitions unless this information is provided to the compiler.

Some compilers and run-time systems provide run-time checking of arguments for type and for the correct number. To do this checking usually requires both the calling program and the called program to be compiled with a special compiler option. Then run-time checking can be done to

determine if the number and types of arguments in function (and subroutine) calls are consistent with the actual function definitions.

MPI (Message Passing Interface) routines are used to pass messages in Fortran, C, and C++ programs running on distributed memory parallel computers. The Fortran *include "mpif.h"* statement does not include interface block information for the MPI routines but the *use mpi* usually does. When this is the case, compile time argument checking can be performed for all MPI routines. The MPI *#include <mpi.h>* statement for C/C++ MPI programs includes the declarators/prototypes for MPI routines so compile time MPI argument checking is also performed by the C and C++ compilers. Cray and SGI have an MPI environment variable, MPI_CHECK_ARGS, which allows a more complete argument checking of MPI routines at run-time than just argument type and the correct number of arguments.

OpenMP directives/pragmas are used to parallelize Fortran, C, and C++ programs running on shared memory parallel computers. The OpenMP Fortran 2.0 API requires a Fortran include file named omp_lib.h or a Fortran 90 module named omp_lib. The API requires the Fortran 90 module to have interface blocks for all OpenMP routines. The OpenMP C/C++ API requires the OpenMP *#include <omp.h>* statement to include the declarators/prototypes for OpenMP routines. Thus, compile time argument checking is provided for the Fortran 90, C, and C++ compilers. Special OpenMP environment variables could provide additional argument checking for OpenMP routines. However, this would not provide much help debugging OpenMP programs since there are few OpenMP routines and it is rare that errors are made in them.

Vendors usually provide special optimized library routines (e.g. IBM's ESSL Library) whose routines can be called from an application program. It is easy to make mistakes in arguments when calling these routines. It is also easy to make mistakes in arguments when calling system routines. It would aid in program debugging if a means to detect and report these errors were provided.

Our view is that when possible it is best to perform argument checking at compile/link-time rather than at run-time since these errors can then be detected and fixed prior to run-time. It would be easier for users to debug their programs if as part of a compiler's general debugging option (e.g. –debug), argument information for library routines and system routines was made available to allow for automatic argument checking at compile time.

4. Debugging programs that generate signals not handled by the language error handler

When executing a program, there are many signals that may be issued by the operating system. These signals are sent to the executing program's error handler. When receiving a signal from the operating system, the program's error handler then deals with each signal in the manner desired by the person who wrote the error handler. For some compilers and run-time systems, the error handler does nothing when receiving certain signals. To aid in debugging the program, the output should be annotated with the signal name and where in the program it occurred

5. Detecting errors with strings at run-time

Some compilers and run-time systems provide Fortran character substring bounds checking at run-time if the program is compiled with a special option.

Since strings are not an intrinsic data type in C but are arrays of characters terminated by a null character, errors in string assignments can be detected by the techniques described in items 6 and 7 listed immediately below. C string assignment is accomplished by element-wise array assignment; by string manipulation functions (e.g. strcpy, strcat) or by I/O functions. Errors arising from array element assignment result in out-of-bounds array index references. Errors arising from string manipulation or I/O functions result in out-of-bounds pointer references.

String bounds checking is automatically performed when using C++ string classes, so C++ string out-of-bounds errors will be detected and handled at run-time without enabling a special compiler option.

6. Detecting Out-of-bounds indexing of statically and dynamically allocated arrays

A common run-time error is the reading and writing of arrays outside of their declared bounds. Some compilers provide special options to detect these errors for Fortran, C, and C++.

7. Detecting Out-of-Bounds Pointer References

A common run-time error for C and C++ programs occurs when a pointer points to memory outside its associated memory block. In the following example, A[5] is not within bounds.

```
float *A;
A=(float *) calloc(5,sizeof(float));
for(i=0;i<5;i++)
     A[i]=i;
A[5]=1.0; /* out-of-bounds writing using subscripts */
```

In the following example, after the for-loop the pointer p points outside the memory block allocated for A.

```
float *A, a, *p;
int i;
A=(float *) calloc(5,sizeof(float));
for(i=0;i<5;i++)
     A[i]=i;
p=A;
for(i=0;i<=5;i++)
  p++;
a=*p; /* out-of-bounds reading using pointers */
```

In the following example, after the for-loop the pointer p points outside the statically allocated array A.

```
float A[5],a,*p;
int i;
for(i=0;i<5;i++)
     A[i]=i;
p=A;
for(i=0;i<3;i++)
  p+=3;
a=*p; /* out-of-bounds reading using pointers */
```

To detect out-of-bounds pointer references requires the logging of all memory allocations, memory deallocations, and all pointer references. With this information, each pointer reference can be checked to determine if it is pointing within its associated memory block. Typically, compilers and run-time systems do not provide this checking. Third-party software is often used to find these run-time errors.

8. Detecting Memory Allocation and Deallocation Errors

Even though the Fortran, C, and C++ standards do not require dynamically allocated memory be deallocated prior to program termination, this does not make efficient use of memory. Memory should be deallocated when it is no longer needed. A memory deallocation error occurs when a portion of memory is deallocated more than once. It is also an error to write a C or C++ program that uses allocation and deallocation constructs in ways where the behavior is undefined or unspecified by the language standard. In the following two examples, the free(a) statement is used incorrectly.

```
int *a, *b;
a = (int *) calloc(32,sizeof(int));
b = (int *) realloc(a, 64*sizeof(int));
free(a);
free(b);

int *a
a = (int *) calloc(32,sizeof(int));
a++;
free(a);
```

Another common source of errors in C and C++ programs is an attempt to use a dangling pointer. A dangling pointer is a pointer to storage that is no longer allocated. These errors are usually hard to find since often the program continues to execute after the error occurs. The following is an example of such an error:

```
char *makeName(int num) {
      char buffer[12];
      sprintf(buffer,"X%d",num);
      return buffer;
}
int main(void)
{
      char *p;
      p=makeName(2);   /* copying dangling pointer */
      printf("p=%s\n",p);   /* reading from a dangling pointer */
      return 0;
}
```

Typically, compilers and run-time systems do not provide this checking with their compilers. Third-party software is often used to find these run-time errors.

9. Detecting File Descriptor Problems

The Fortran, C, and C++ standards do not require files opened during a program's execution be closed prior to the program's termination. Files that have been opened but not closed will be closed automatically when the program terminates. Failing to close files that are no longer needed does not cause a problem unless the program aborts during execution due to the file descriptor limit being exceeded. File descriptors are also consumed by inter-processor communication, e.g. when using sockets. The problem of exceeding the file descriptor limit can be fixed by either increasing this limit and/or by closing files when they are no longer needed. It is good programming practice to close files when they are no longer needed. To find files that have been opened but not closed prior to program termination, one can log when files are opened and when they are closed. All files that have not been closed prior to program termination could then be reported.

Multiple opening and closing of the same file may also cause problems. The Fortran standard allows the *close* statement to refer to a unit that is not connected or does not exist. It is also valid to open a file that has already been opened. However, Fortran does not allow connecting two different units to the same file.

The C and C++ standards do not specify if it is legal to open a file that is already open, nor if it is legal to close a file that has not been opened. We consider the issuing of extra opens and closes to be poor programming practice and recommend that a warning message be issued at run-time.

When running parallel programs one might want to open/close the same file from different threads/processes. Thus, when executing parallel programs one would like a warning message issued only when a file is being opened/closed from the same thread or process that already opened/closed the file.

10. Detecting Memory Leaks

A program has a *memory leak* if during execution the program loses its ability to address a portion of memory because of a programming error; as a result the program cannot deallocate or reallocate for future use this portion of memory. If memory leaks occur frequently enough, this may cause the program or operating system to crash. The following lists some examples where memory leaks can occur in Fortran, C, and C++.

- A pointer points to a location in memory and then all the pointers pointing to this location are set to point somewhere else. It is then impossible to use this memory during the remainder of the program's execution.
- A function/subroutine is called, memory is allocated during execution of the function/subroutine, and then the memory is not deallocated upon exit and all pointers to this memory are destroyed.

```
func()  {
    int *p;
    p = (int *) calloc(5,sizeof(int))
}
```

- In C++ programs a memory leak might occur when an exception is being handled. In the following example the destructor of class A will never be called resulting in a memory leak:

```
class A {
public:
```

```
    A() {
       dummy = new char[100];
       throw "boom"; }
    ~A() { delete dummy; }
    char *dummy;
};
int main()  {
    try {  A *pa = new A();  }
    catch(...) { }
    return 0;
}
```

To detect memory leaks at run-time requires the logging of all memory allocations, memory deallocations, and all pointer references.

11. Fortran 90 array conformance checking
Fortran compilers should provide run-time checking of Fortran 90 array conformance.

III. Software for Detecting Run-Time Errors

The following software products are designed to find run-time errors in C, and/or C++, and/or Fortran. They were selected on the basis of performing a number of web searches. The information about the various software products listed below was obtained primarily from documentation on their web sites. Of course, claims on web sites should be verified for accuracy. The following is a summary of what was found for each software product. The software products have been divided into commercial and non-commercial categories.

III.1 Commercial Software

C.1 Great Circle from Geodesic Systems

http://www.geodesic.com/solutions/greatcircle.html

General Information

This is a commercial product designed to detect the following run-time errors:

- buffer overwrite
- memory leaks
- multiple frees

Languages Supported: C and C++
Platforms supported: AIX, HP-UX, Linux, Solaris, Windows

C.2 Insure++, by ParaSoft

http://www.parasoft.com/products/insure/index.htm

General Information

Insure++ is a commercial product from ParaSoft Corporation. Insure ++ is an automatic run-time application testing tool that detects errors such as memory corruption, memory leaks, memory allocation errors, variable initialization errors, variable definition conflicts, pointer errors, library errors, logic errors, and some algorithmic errors. Insure++ indicates where errors are made in the

source code (Insure++ instruments source code) and also provides coverage analysis. Insure++ uses a patented "mutation testing" technique to help find program errors automatically. It detects the following run-time errors:

- memory corruption due to reading or writing beyond the valid areas of global, local, shared, and dynamically allocated objects.
- operations on uninitialized, NULL, or "wild" pointers.
- memory leaks.
- errors allocating and freeing dynamic memory.
- string manipulation errors.
- operations on pointers to unrelated data blocks.
- invalid pointer operations.
- incompatible variable declarations.
- mismatched variable types in printf and scanf argument lists.
- unused variables
- invalid parameter passing

Languages Supported: C and C++
Platforms supported: Windows, AIX, Linux, Solaris, HPUX

C.3 ObjectCenter, by Centerline

http://www.centerline.com/productline/object_center/object_center.html

General Information

Centerline's ObjectCenter is a comprehensive debugging tool that extends their TestCenter product to C++. It is not clear from the online documentation if there is any functionality in ObjectCenter for detecting run-time errors that is not in their TestCenter product. Source code is instrumented at compile time. This is commercial software that detects the following run-time errors:

- memory leaks
- duplicate frees
- illegal access errors on the heap, stack, and in static memory
- reads/stores past the end of an array, inside structs, and inside classes
- illegal cast and downcast errors
- inconsistent cross-module definitions

Languages Supported: C and C++
Platforms supported: HPUX, Solaris

C.4 Purify, by Rational Software (now owned by IBM)

http://www.rational.com/products/purify_unix/index.jsp

General Information

Purify is a commercial product sold by Rational Software that detects a wide variety of program errors in C and C++ programs. IBM has recently purchased Rational Software, see www.rational.com. Purify modifies the program's object files when they are linked together. Rational Software claims that Purify: (1) has long been the standard in error detection for Sun, HP, and SGI UNIX platforms, and (2) provides the most complete error and memory leak detection available. Purify checks all application codes, including source code, third party libraries, and shared and system libraries. The following run-time errors are detected:

- memory leaks

- duplicate frees
- illegal access errors

Languages Supported: Fortran, C, and C++
Platforms supported: Solaris, HPUX, IRIX

C.5 QC, by Onyx Technology

http://www.onyxtech.com/products/qc.html

General Information

QC is commercial software that stress-tests applications during runtime. In particular, QC is specifically designed to make memory related bugs reproducible. It has been designed to be integrated into the Mac system software.

C.6 SmartHeap, by MicroQuill

http://www.microquill.com/smartheap/index.html

General Information

SmartHeap is a commercial product that provides run-time detection of heap memory error. SmartHeap-SMP is a threaded, parallel version of SmartHeap for faster analysis of C and C++ programs. It detects the following run-time errors:

- memory overwrites before or after an allocated block..
- memory overwrites over any internal heap data structures.
- memory leaks.
- invalid function parameters.
- double freeing, writes, or references to previously deallocated memory.
- writes into memory that has been deallocated.
- freeing/reallocating a block marked as "no-free" or "no-realloc" .

Languages Supported: C and C++
Platforms supported: Windows, AIX**,** Linux, Solaris, Irix**,** HPUX

C.7 TestCenter, by Centerline

http://www.centerline.com/productline/test_center/test_center.html

General Information

CenterLine's TestCenter is a commercial product designed to find memory leaks and other run-time errors. Currently, TestCenter only run on Solaris, AIX, and HPUX platforms. It detects the following run-time errors:

- pointer and array bounds violation
- unaligned pointer errors
- using unallocated memory
- improper use of malloc/free
- run-time error checking in many C library functions
- unanticipated termination signals
- allocated memory that has not been freed and has no pointer to it
- memory allocated but not freed

Languages Supported: C and C++
Platforms supported: Solaris, HPUX, AIX

C.8 ZeroFault, by The Kernel Group

http://www.zerofault.com/

General Information

ZeroFault is a commercial product from The Kernel Group that finds memory errors in Fortran, C and C++ programs. However, this product currently only runs on IBM's AIX machines. It detects the following run-time errors:

- memory leaks
- reads or writes of unallocated stack, heap, and static memory
- reads of uninitialized stack, heap, and static memory
- reads or writes of freed heap memory
- attempts to free or realloc unallocated memory
- invalid arguments passed to system calls and common functions

Languages Supported: Fortran, C, and C++.

Platforms supported: AIX

III.2 Non-Commercial Software

NC.1 Ccmalloc, by Armin Biere

http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/

General Information

This non-commerial software has been developed to detect memory leaks for C and C++ programs. The documentation on its web site states: "It does not replace more sophisticated tools such as *purify* or *valgrind*, since it can not detect illegal memory reads, at least not all type of memory reads." Ccmalloc does not require recompiling since it links with the executable. The generated log files can be very large. Distributed with a GPL (General Public License – http://www.gnu.org/licenses/gpl.txt) so source code is freely available. Ccmalloc finds the following run-time errors:

- memory leaks
- multiple deallocation of the same data
- under writes and over writes
- writes to already deallocated data

Languages Supported: C and C++

Platforms supported: Linux, Solaris, FreeBSD

NC.2 CSRI UToronto Malloc, by Mark Moraes

ftp://ftp.cs.toronto.edu/pub/moraes/malloc.tar.gz, Mark Moraes, moraes@deshaw.com.

General Information

This is free software to detect the run-time errors listed below for C programs. It is written in ANSI C so it should run on any platform with a C compiler that supports ANSI C. For more information, see http://www.cs.colorado.edu/~zorn/Malloc.html#csrimalloc . The following errors are detected:

- memory leaks
- heap corruption

Languages Supported: C

Platforms supported: any machine with an ANSI C compiler

NC.3 Debug Malloc Library, by Gray Watson

http://www.dmalloc.com/

General Information

This is free software for debugging memory allocation/deallocation errors and has been designed as in replacement for the system's `malloc`, `realloc`, `calloc`, `free` and other memory management routines while providing debugging facilities that are configurable at runtime. The following run-time errors are detected:

- memory-leaks;
- fence-post write detection;

Languages Supported: C and C++

Platforms supported: AIX, BSD/OS, DG/UX, Free/Net/OpenBSD, HPUX, Irix, Linux, Solaris, Windows, Cray T3E

NC.4 Memdebug, by Rene Schmit

http://www.netsw.org/system/libs/memory/debug/

General Information

Memdebug is free software that detects the errors listed below. Information about Memdebug is found in the READ.ME file found at the above URL. Additional information can be found at http://www.cs.colorado.edu/homes/zorn/public_html/MallocDebug.html#memdebug.The following run-time errors are detected:

- memory leaks
- duplicate frees
- freeing an illegal pointer
- out-of-bounds pointer references
- off-by-one errors in memory blocks
- using an unallocated pointer (a pointer is used that was not obtained by a call to malloc(), calloc() or realloc() in the realloc() and free() routines)

Languages Supported: C

Platforms supported: HPUX, Solaris, True64, Windows, MacOS

NC.5 Mpatrol, by Graeme Roy

http://www.cbmamiga.demon.co.uk/mpatrol/

General Information

Mpatrol is a freely-available (with a GPL License), memory allocation library for C and C++ that provides extensive debugging, profiling and tracing capabilities to help fix memory allocation errors and monitor memory allocation behavior. It can detect heap corruption using overflow buffers, virtual memory protection or software watchpoints on supported systems, and can also help pinpoint memory leaks with their associated symbolic stack tracebacks. mpatrol comes with a suite of utility programs, is supported with extensive documentation and works with numerous platforms including most varieties of UNIX, Linux and Windows. For additional information, see http://www.cbmamiga.demon.co.uk/mpatrol/mpatrol.html#SEC_Top. The following run-time errors are detected:

- memory leaks

- memory allocation problems
- heap corruption

Languages Supported: C and C++
Platforms supported: Red Hat Linux, True64, Free BSD, AIX, HPUX, IRIX, Solaris, Windows NT

NC.6 Mprof, by Ben Zorn

http://web.mit.edu/afs/athena/contrib/watchmaker/src/mprof-3.0/, Ben Zorn, zorn@cs.colorado.edu
General Information
Mprof is free software that profiles dynamic memory allocations for C programs. Mprof is a two-phase tool. The monitor phase is linked into executing programs and records information each time memory is allocated. The display phase reduces the data generated by the monitor and presents the information to a user in four tables: a list of memory leaks, an allocation bin table, a direct allocation table, and a dynamic call graph. Mprof detects the following run-time errors:

- memory leaks
- heap corruption

Languages Supported: C
Platforms supported: Solaris, Irix

NC.7 MSS (Memory Supervision System)

http://hem.passagen.se/blizzar/mss/
General Information:
This is free (GPL) software designed to find bug in C and C++ programs caused by the misuse of dynamically allocated memory. It will run on any platform with ANSI C and C++ compilers. MSS will detect the following run-time errors:

- memory leaks
- use of uninitialized memory
- zero-length allocations
- out of range block accesses
- bogus or repeated deallocations
- unsuccessful allocations
- corrupted pointers

Languages Supported: C and C++
Platforms supported: all platforms supporting ANSI C and C++ compilers

NC.8 Valgrind

http://valgrind.org
General Information
Valgrind is a freely-available (with a GPL License) software product that helps one find memory management problems in C and C++ programs. Valgrind works directly with the executables, with no need to recompile, relink, or modify the program to be checked. Valgrind requires a "complete CPU simulation" in software. Valgrind detects the following run-time errors:

- use of uninitialized memory
- reading/writing memory after it has been free'd
- reading/writing off the end of malloc'd blocks

- reading/writing inappropriate areas on the stack
- memory leaks -- where pointers to malloc'd blocks are lost forever
- passing of uninitialized and/or unaddressible memory to system calls
- mismatched use of malloc/new/new[ ] versus free/delete/delete[ ]

Languages Supported: Fortran, C and C++
Platforms supported: X86/Linux, AMD64/Linux, and PPC32/Linux

NC.9 Electric Fence, by Bruce Perens

http://perens.com/FreeSoftware/ElectricFence/

General Information

Electric Fence is a freely available (GPL) and easy to use malloc() debugger, which can be used simply by placing its library in the appropriate position in an application's shared-library path. Electric Fence is a different kind of malloc() debugger. It uses the virtual memory hardware of your system to detect when software overruns the boundaries of a malloc() buffer. It will also detect any accesses of memory that has been released by free(). Electric Fence detects the following run-time errors:

- Buffer overruns /underruns
- Illegal memory accesses

Languages Supported: C
Platforms Supported: Linux, Solaris, HP/UX, and OSF

Tables 1 and 2 summarize much of the information presented in this section and are based on reading documentation and not on actual running test programs.

	Languages Supported	**Methods Used for Error Detection**	**Platforms Supported**	**Availability for Evaluation**
Commercial software				
Great Circle	C / C++	unknown	AIX, HP-UX, Linux, Solaris,	60 days
Insure++	C / C++	Instrumenting the source code	Windows, AIX, Linux, Solaris, HPUX	free evaluation copy
ObjectCenter	C / C++	Instrumenting the source code	HPUX, Solaris	no response to email
Purify	C / C++	Modifying object files	HPUX, IRIX, Solaris	7 days evaluation copy
SmartHeap	C / C++	Malloc() replacement	Windows, AIX**,** Linux, Solaris, Irix**,** HPUX	demo available
TestCenter	C / C++	Modifying object files	Solaris, HPUX, AIX	no response to email
ZeroFault	C / C++ Fortran	Dedicated memory debugger (works with executables)	AIX only	evaluation copy for small programs
Non-commercial software				
Ccmalloc	C / C++	Malloc() replacement	all with ANSI C	GPL
CSRI UToronto Malloc	C	Malloc() replacement	all with ANSI C	GPL
Dmalloc	C / C++	Malloc() replacement	all with ANSI C	Free
Memdebug	C	Malloc() replacement	all with ANSI C	Free
Mpatrol	C / C++	Malloc() replacement	all with ANSI C	GPL
Mprof	C	Malloc() replacement	all with ANSI C	Free
MSS	C / C++	Malloc() replacement	all with ANSI C	GPL
Valgrind	C / C++ Fortran	Works with executables	X86/Linux, AMD64/Linux, PPC32/Linux	GPL

Table 1. Summary of general information about the software products.

Products	Type of Memory Checked	Out-of-Bounds Pointer and Index References				Memory Allocation & Deallocation Errors	Memory Leaks
		Off by one (fence post error)		Off by any n			
		Writing	Reading	Writing	Reading		
Commercial software							
Great Circle	heap	+	–	–	–	+	+
Insure++	heap&stack	+	+	+	+	+	+
ObjectCenter	heap&stack	+	+	?	?	+	+
Purify	heap&stack	+	+	+	+	+	+
SmartHeap	heap	+	–	≤ fixed n	–	+	+
TestCenter	heap	+	–	–	–	+	+
ZeroFault	heap&stack	+	+	+	+	+	+
Non-commercial software							
Ccmalloc	heap	+	–	≤ fixed n	–	+	+
CSRI UToronto Malloc	heap	+	–	–	–	not all	+
Dmalloc	heap	+	–	–	–	+	+
Memdebug	heap	+	–	–	–	+	+
Mpatrol	heap	+	+	≤ fixed n	–	+	+
Mprof	heap	-	–	–	–	+	+
MSS	heap	+	–	≤ fixed n	–	+	+
Valgrind	heap&stack	+	+	+	+	+	+

Table 2. Summary of error detection for each product from product documentation.

+	the software can detect these kinds of errors according to documentation
-	the software cannot detect this kind of errors according to documentation
≤ fixed n	checks out-of-bounds references when off by any i, where $1 \leq i \leq n$ and where n is a fixed integer specified via an option in the error detection software
?	not clear from documentation

Table 3. Symbols used for Table 2.

Based on the information gathered in this section, Insure++, Purify, Ccmalloc, Memdebug, Mpatrol, MSS, and Valgrind were evaluated running the tests written for this study to determine how well they could detect various run-time errors.

IV. Software Evaluation Methodology

A variety of run-time error tests were written for each of run-time errors listed above in section II that apply to C and C++ programs. Tables 5 and 6 show the results of running these tests. The run-time error tests that have been written are simple tests to check for the software product's ability to detect errors in simple situations. These tests were run using the GNU C and C++ compilers.

When a software product is able to detect run-time errors in relatively simply written tests, this does not necessarily mean it will be able to detect the same errors in large application codes since internal tables could overflow or other buffers might run out of space. Thus, testing included using the various software products on two "large" C codes:

1. MILC, a lattice gauge theory code (using the dynamical Kogut-Susskind fermion simulation) with about 40,000 lines of code, see http://www.physics.utah.edu/~detar/milc/.
2. Ftnchek (version 3.2.2), a static analyzer for Fortran 77 programs with about 33,000 lines of code, see http://www.dsm.fordham.edu/~ftnchek/ftp-index.html.

The software products were also evaluated for ease-of-use. Ease-of-use includes whether the software issues an appropriate message and if a traceback to where the problem occurred is provided.

V. Results of running the C and C++ run-time error detection tests

Table 4 summarizes our experiences with using each product. Tables 5 and 6 summarize the results of running our test suites with the different software products for both C and C++, respectively. The first number in each box indicates the number of tests passed and the second number indicates the number of tests in this category. For example, 4/7 means the software correctly detected errors in 4 of the 7 tests.

Software	How to use	Comments
Ccmalloc	Changes to source code are not needed. Must link with ccmalloc libraries.	Easy to use. Reports line numbers.
Memdebug	One must either recompile after adding #include “memdebug.h” in source or use the -include gcc compiler option.	Easy to use. Good error messages. Reports line numbers.
Mpatrol	One must either recompile after adding #include “mpatrol.h” in source or use the -include gcc compiler option, and then link with mpatrol libraries.	Not easy to use due to too many options that need to be set. Errors messages are sometimes not clear. Reports line numbers.
Mss	One must either recompile after adding #include “mss.h” in source or use the -include gcc compiler option, and then link with mss libraries.	Easy to use. Reports line numbers.
Valgrind	Works with executables but recompilation with –g option is required to obtain line number information.	Very easy to use. Very good error messages. Reports line numbers.
Insure++	Instruments source code. Thus, recompilation is required.	Extremely easy to use. Excellent error messages. Reports line numbers. Very easy to fix errors in source from Insure Window. Performed best.
Purify	Instruments object code Requires recompilation with –g option to obtain line number information.	Extremely easy to use. Usually gives excellent error messages. Reports line numbers. Very easy to fix errors in source from Purify Window. Performed second best.

Table 4. Evaluation summary.

	Ccmalloc	Memdebug	Mpatrol	MSS	Valgrind	Insure++	Purify
A. Stack Memory Access Errors							
Out-of-bound reading/writing using subscripts	0/19	0/19	0/19	0/19	0/19	19/19	12/19
Out-of-bound reading/writing using pointers	0/8	0/8	0/8	0/8	0/8	8/8	2/8
Writing over a function	0/2	0/2	2/2	0/2	0/2	2/2	2/2
Using uninitialized variables	0/2	0/2	0/2	0/2	0/2	2/2	0/2
Reading/writing data without proper alignment	0/4	0/4	4/4	0/4	0/4	4/4	4/4
Reading/writing using null pointers	0/4	0/4	4/4	0/4	4/4	4/4	4/4
Using non-initialized pointers	0/2	0/2	0/2	0/2	2/2	2/2	2/2
B. Heap Memory Access Error							
Out-of-bound reading using subscripts	0/11	0/11	9/11	0/11	11/11	11/11	10/11
Out-of-bound writing using subscripts	0/11	4/11	11/11	4/11	11/11	11/11	10/11
Out-of-bound reading using pointers	0/8	0/8	6/8	0/8	8/8	8/8	8/8
Out-of-bound writing using pointers	0/8	4/8	8/8	2/8	8/8	8/8	7/8
Reading/writing freed memory using subscripts	4/8	0/8	6/8	0/8	8/8	8/8	8/8
Reading/writing freed memory using a pointer	4/8	0/8	0/8	0/8	8/8	8/8	7/8
Using unallocated memory	0/2	0/2	2/2	0/2	0/2	2/2	2/2
Using uninitialized variables	0/2	0/2	0/2	0/2	0/2	1/2	0/2
Allocate memory beyond memory available to program	2/6	6/6	4/6	1/6	3/6	6/6	6/6
Reading/writing using null pointers	2/3	3/3	3/3	3/3	0/3	3/3	0/3
Storing to both elements in a union	0/2	2/2	2/2	2/2	2/2	2/2	2/2
Reading/writing data without proper alignment	0/6	3/6	6/6	2/6	2/6	6/6	6/6
Using null pointers in realloc	1/1	1/1	1/1	0/1	1/1	1/1	1/1
C. Overflow/Underflow/Divide by zero	0/7	0/7	0/7	0/7	0/7	2/7	2/7
D. Memory Leaks	19/19	19/19	19/19	19/19	19/19	19/19	19/19
E. File Descriptor Leaks	10/10	0/10	10/10	0/10	10/10	10/10	0/10
F. File Descriptor Errors	2/8	0/8	3/8	0/8	8/8	6/8	4/8
G. Call to standard functions/system calls with incorrect parameters	0/17	4/17	4/17	0/17	5/17	8/17	12/17
H. Allocation/Deallocation Errors	16/16	16/16	16/16	15/16	16/16	16/16	14/16
I. Miscellaneous	0/4	1/4	1/4	0/4	1/4	4/4	1/4
J. Signals	0/27	0/27	0/27	0/27	0/27	11/27	15/27
K. Multi-Files Program Errors	7/19	7/19	11/19	0/19	10/19	19/19	16/19

Table 5. Results of running C tests.

	Ccmalloc	Mpatrol	MSS	Valgrind	Insure++	Purify
Memory Leaks	19/20	20/20	20/20	19/20	20/20	20/20
Memory Access Errors						
Out-of-bound reading using subscripts – stack memory	0/8	0/8	0/8	4/8	8/8	1/8
Out-of-bound writing using subscripts – stack memory	0/8	0/8	0/8	4/8	8/8	2/8
Out-of-bound reading using pointers – stack memory	0/4	0/4	0/4	1/4	4/4	0/4
Out-of-bound writing using pointers – stack memory	0/4	0/4	0/4	1/4	4/4	0/4
Out-of-bound reading – dynamic memory	0/26	22/26	0/26	26/26	26/26	22/26
Out-of-bound writing – dynamic memory	0/29	25/29	11/29	29/29	29/29	25/29
Others	8/10	9/10	3/10	5/10	10/10	10/10
File Descriptor Leaks	10/10	10/10	0/10	0/10	10/10	10/10
Using uninitialized variables	0/5	0/5	0/5	2/5	3/5	5/5
Throw Exception	0/5	5/5	0/5	0/5	5/5	0/5
Memory Allocation/Deallocation Errors	24/27	26/27	19/27	8/27	26/27	26/27
Miscellaneous	2/4	2/4	1/4	3/4	4/4	2/4
Multi-Files Program Errors	12/12	12/12	0/12	12/12	12/12	10/12

Table 6. Results of running C++ tests.

VI. Comments based on testing the selected software products

This section contains our comments for each of the software products that we selected to run the tests against. All tests were run using the GNU C and C++ compilers. All software products were run with two large codes MILC and ftnchek, see Section IV for more information. All of the selected software products were able to run MILC and ftnchek without any problems.

CCMALLOC is a memory profiling and malloc debugging library for C and C++ programs that requires the GNU debugger, gdb. All tests were run using the GNU C and C++ compilers. We found CCMALLOC to be easy to install. It needs only to be linked with the object code of an application so no recompilation of the application is needed. CCMALLOC requires that different options be set in order to detect different kinds of errors. We consider the documentation for CCMALLOC to be poorly written and often difficult to understand (the file *ccmalloc.cfg* is the only available user manual). Poor documentation made the product difficult to use. CCMALLOC was not able to detect "writing beyond the bounds of the allocated memory block" errors. The person who developed CCMALLOC, Armin Biere, was contacted. He confirmed that this is a bug but did not have time to fix it immediately.

For multiple deallocations of the same memory block and writes to previously deallocated memory blocks, CCMALLOC didn't indicate the line numbers where these errors occurred.

In conclusion, we found CCMALLOC good at detecting memory leak problems but poor at detecting many other run-time errors. Poor documentation made CCMALLOC difficult to use.

MSS is free (GPL) software designed to find errors in C and C++ programs caused by the misuse of dynamically allocated memory. We found MSS to be easy to install. MSS requires either the statement *#include "mss.h"* be added to each source file before compiling or one can simply use the *–include* option on the GNU compiler to have the *#include "mss.h"* automatically added to each source file. MSS also requires the insertion of *#define MSS* into each source file **or** one can use the *–D* compiler option with the gcc compiler and not have to modify source code.

MSS also provides some special macros to help in the debugging process, e.g.

- MSS_CHECK_BLOCK_AT(ptr) - checks the specified block of memory starting at 'ptr' for out-of-bound writes.
- MSS_CHECK_POINTER_VALIDITY(ptr) – checks if the specified pointer 'ptr' points anywhere within a legal block of memory.

It is useful to have these macros but sometimes it is difficult to know where to insert these macros into the source code. MSS was not able to detect writes to previously deallocated memory blocks.

In conclusion, we found MSS to be good at finding memory leaks and allocation/deallocation errors and poor at detecting other run-time errors. MSS is easy to use with the GNU C/C++ compilers.

MPATROL is a debugging tool for detecting run-time errors that are caused by the misuse of dynamically allocated memory for C and C++ programs. MPATROL requires either the statement *#include "mpatrol.h"* be added to each source file before compiling or one can simply use the –

include option on the GNU compiler to have the *#include "mpatrol.h"* automatically added to each source file.

MPATROL has many options. We found that sometimes it was difficult to decide what options to set when running our various tests. MPATROL does have reasonably well-written documentation.

For all out-of-bounds errors for dynamically allocated memory, MPATROL showed the line number of the allocation call instead of the line number where the incorrect reading/writing occurred. MPATROL detected out-of-bounds reading for heap memory but only for subscripts less than 5 below the allocated array index and for any subscript number greater than size of the memory block. Some error messages were not clear.

In conclusion, MPATROL did well on most, but not all, of our C and C++ tests. We found that sometimes it was difficult to decide what options to set when running our various tests.

MEMDEBUG is a debugging tool for the run-time detection of the following errors in C (and not C++) programs: memory leaks, duplicate deallocations, deallocating an illegal pointer, out-of-bounds pointer references, off-by-one errors in memory blocks, and unallocated pointer. We found MEMDEBUG to be easy to install and to use. MEMDEBUG requires the addition of *#include "memdebug.h"* to each source file before compiling or one can simply use the *–include memdebug.h* option on the GNU compiler to have the *#include "memdebug.h"* automatically added to each source file. MEMDEBUG also requires the insertion of *#define MEMDEBUG* into each source file **or** one can use the *–D* compiler option with the gcc compiler and not have to modify source code.

Memdebug was not able to detect general out-of-bounds errors but only off-by-one.

In conclusion, MEMDEBUG did well on our memory leaks, and allocation/deallocation tests, but did not do well on the other tests. MEMDEBUG is easy to use with the GNU C compiler.

VALGRIND is a freely-available (with a GPL License) software product for detecting memory management problems at run-time. The GNU C and C++ compilers were used for testing. Valgrind works directly with the executables, with no need to recompile, relink or modify the program to be checked. To be able to report line number information, Valgrind does require source code be compiled with the *–g* option.

We found Valgrind to be easy to install, easy to use, and performed well on most, but not all, of our tests. Valgrind reports not only the location where the error happened, but also where the associated memory block was allocated/deallocated.

In conclusion, Valgrind did well on most (but not all) of our tests.

Insure++ is a commercial software product that instruments source code for detecting run-time errors in C and C++ programs. We found Insure++ to be easy to install and very easy-to-use. For example, instrumentation and compilation is accomplished by issuing *insure gcc -g* instead of the normal compilation command *gcc*. It is especially easy to correct errors since Insure++ has a

special window that shows the source code in the neighborhood of where each error occurred so one can easily correct program errors. Insure++ also has a special window that lists the percentages of lines of code executed. (Insure++ also shows which lines were executed.) This is useful information to someone debugging a program. We found the documentation to be thorough and clearly written. We consider Insure++ to be the best of all the software products evaluated.

Purify is a commercial software product that uses existing C and C++ object code and does not require the instrumentation of source code. To obtain line number information, Purify requires compilation with the *–g* option. Having to recompile source code with the –g option to obtain line number information takes away much of the advantage of Purify's ability to find run-time errors without modifying source code. Like Insure++, Purify also has a special window that shows the source code in the neighborhood of where each error occurred so one can easily correct program errors. Like Insure++, Purify also provides code coverage analysis.

We consider Purify to be the second best product of all the software products evaluated. The Conclusions section contains detailed comparisons between Insure ++ and Purify.

VII. Conclusions

Both commercial products, Insure++ and Purify, did the best of all the software products we evaluated on our run-time error function tests. Insure++ requires the instrumentation of source code and then compiling. We found this process easy to do. Purify does not require the instrumentation of source code but does require recompilation with the *–g* option to be able to report line numbers in the source code. It is our opinion that having line number information is essential to the debugging process.

Error messages were usually better and clearer when using Insure++ than when using Purify. For example, for the throw exception tests, Purify only reports a core dump occurred and does not report a line number. However, Insure++ detects the error and gives the line number where the error occurred.

Another example illustrating that Insure++ provides better and clearer error messages occurred when the same pointer was used in two malloc's in a way that created a memory leak. For this situation Insure++

- gives the name of this pointer variable and line numbers where the first and second allocations occurred, and
- gives the cause of the memory leak: "Memory leaked due to pointer reassignment".

However, Purify only reports that a memory leak occurred and gives the line number of the first allocation. Additionally, unlike Insure++, Purify incorrectly reported unfreed memory that was addressable until the end of the program as a memory leak.

Performance should also be considered when comparing Insure++ and Purify. To get an idea of the performance differences, we ran the MILC code on a Sun Ultra 10 using both Insure++ and Purify and compared compile times and execution times. Remember that Purify requires compilation with the *–g* option, so the Purify times are the compile times with this option. These times were compared when compiling MILC using the same compiler, gcc, with the *–g* option and gcc without

the *–g* option. Four runs were made. Table 7 summarizes the wall-clock times measured with times in seconds and shows minimum, maximum, and average times.

	Insure++	Purify	gcc -g	gcc
min compile time	33.8 secs	7.7 secs	6.1 secs	5.2 secs
max compile time	34.4 secs	8.1 secs	6.1 secs	5.3 secs
average compile time	34.1 secs	7.9 secs	6.1 secs	5.2secs
min run time	94.8 secs	120.3 secs	4.1 secs	4.0 secs
max run time	96.0 secs	122.0 secs	4.3 secs	4.1 secs
average run time	95.2 secs	121.0 secs	4.2 secs	4.1 secs
average compile time + run time	129.3 secs	128.9 secs	10.3 secs	9.3 secs

Table 7. Timing comparisons in seconds for MILC running on a Sun Ultra 10.

Compile times are about 4 times slower when using Insure++ than when using Purify. Insure++ execution times are about 23 times slower than when using gcc alone. Purify execution times are about 29 times slower than when using gcc alone. When debugging a program, one usually makes a change to the source code, compiles, and then re-runs. Notice that the total time when using Insure++ is nearly the same as when using Purify for the MILC application.

Our evaluation shows that the overall capability of detecting run-time errors of the non-commercial products is significantly lower than the quality of both Purify and Insure++. We also found Purify and Insure++ to be much easier to use than the non-commercial products.

Of all the non-commercial products, Mpatrol provided the best overall capability to detect run-time errors in C and C++ programs. We did find Mpatrol to be difficult to use because it was sometimes not clear what options needed to be set to be able to detect differing run-time errors. We also found some of the error messages produced by Mpatrol to be unclear.