
Debugging and Tuning

Linux for EDA

Fabio Somenzi

Fabio@Colorado.EDU

University of Colorado at Boulder

Outline

- Compiling
 - gcc
 - icc/ecc
 - Debugging
 - valgrind
 - purify
 - ddd
 - Profiling
 - gcov, gprof
 - quantify
 - vtl
 - valgrind
-

Compiling

- Compiler options related to
 - static checks
 - debugging
 - optimization
- Profiling-driven optimization

Compiling with GCC

- `gcc -Wall -O3 -g`
 - reports most uses of potentially uninitialized variables
 - `-O3` (or `-O6`) necessary to trigger dataflow analysis
 - can be fooled by

```
        if (cond) x = VALUE;
        ...
        if (cond) y = x;
```
 - Uninitialized variables not considered for register allocation may escape
 - Achieving `-Wall`-clean code is not too painful and highly desirable
 - Compiling C code with `g++` is more painful, but has its rewards
-

Compiling with GCC

- `gcc -mcpu=pentium4 -malign-double`
 - `-mcpu=pentium4` optimizes for the Pentium 4, but produces code that runs on any x86
 - `-march=pentium4` uses Pentium 4-specific instructions
 - `-malign-double` forces alignment of double's to double-word boundary
 - Use either for all files or for none
 - `gcc -mfpmath=sse`
 - Controls the use of SSE instructions for floating point
 - For complete listing, check gcc's info page under
 - Invoking gcc → Submodel Options
-

Compiling with ICC

- ICC is the Intel compiler for IA-32 systems.
 - <http://www.intel.com/software/products/>
 - `icc -O3 -g -ansi -w2 -Wall`
 - Aggressive optimization
 - Retain debugging info
 - Strict ANSI conformance
 - Display remarks, warnings, and errors
 - Enable all warnings
 - Remarks tend to be a bit overwhelming
 - Fine grain control over diagnostic: see man page
-

Compiling with ICC

- `icc -tpp7`
 - Optimize instruction scheduling for Pentium 4
 - Also `icc -mcpu=pentium4`
 - `icc -ipo`
 - Multi-file interprocedural optimizations
 - `icc -axW`
 - Generate both Pentium 4 and generic instructions
 - `icc -xW`
 - Generate code specific for the Pentium 4
 - Also `icc -march=pentium4`
 - `icc -align`
 - Analyze and reorder memory layout
-

GCC: Profiler-Driven Optimization

- `gcc -fprofile-arcs test.c`
 - Instrumented compilation
 - `./test input`
 - Instrumented execution
 - Produces `.da` files
 - Can be repeated with different inputs
 - `gcc -fbranch-probabilities test.c`
 - Feedback compilation
-

ICC: Profiler-Driven Optimization

- `icc -prof_gen test.c`
 - Instrumented compilation
 - `./test input`
 - Instrumented execution
 - Produces `.dyn` and `.dpi` files
 - Can be repeated with different inputs
 - `icc -prof_use test.c`
 - Feedback compilation
-

Debugging

- Dynamic analysis tools
 - valgrind, purify
- Classical debuggers
 - gdb, idb and their graphical front-ends, especially...
 - ddd

Valgrind

- Tool for debugging and profiling Linux-x86 executables
 - Valgrind consists of:
 - **core**: synthetic CPU
 - **skins**: perform analyses
 - Available skins
 - memcheck and addcheck: memory debugging
 - cachegrind: cache profiling
 - helgrind: races in multithreaded programs
-

Valgrind: Memory Debugging

- 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 unaddressable memory to system calls
 - Mismatched use of malloc/new/new [] vs. free/delete/delete []
 - Some misuses of the POSIX pthreads API
-

Valgrind: Memory Debugging

```
1: #include <stdlib.h>
2: main()
3: {
4:     char *x, *d = "foo";
5:
6:     x = malloc(922);
7:     x = malloc(123);
8:     x = malloc(-9);
9:
10:    free(d);
11:    free(x);
12:    free(x);
13: }
```

Valgrind: Memory Debugging

```
valgrind -leak-check=yes -show-reachable=yes mtest
```

- Warning: silly arg (-9) to malloc()
- Invalid free() / delete / delete[]
 - in main (mtest.c:10)
- 123 bytes in 1 blocks are definitely lost
 - in main (mtest.c:7)
- 922 bytes in 1 blocks are definitely lost
 - in main (mtest.c:6)

Why isn't the double `free(x)` reported?

Valgrind: Memory Debugging

- Valgrind tracks each byte with nine status bits
 - one tracks addressability of that byte
 - the other eight track the validity of the byte
 - Valgrind can be used to debug dynamically-linked ELF x86 executables, without modification, or recompilation
 - `valgrind ls -ls`
 - Valgrind can attach GDB to the running program at the point(s) where errors are detected
 - Valgrind works on large applications
 - Mozilla
 - emacs-21.2
 - AbiWord
 - OpenOffice
 - Gcc
 - KDE3
-

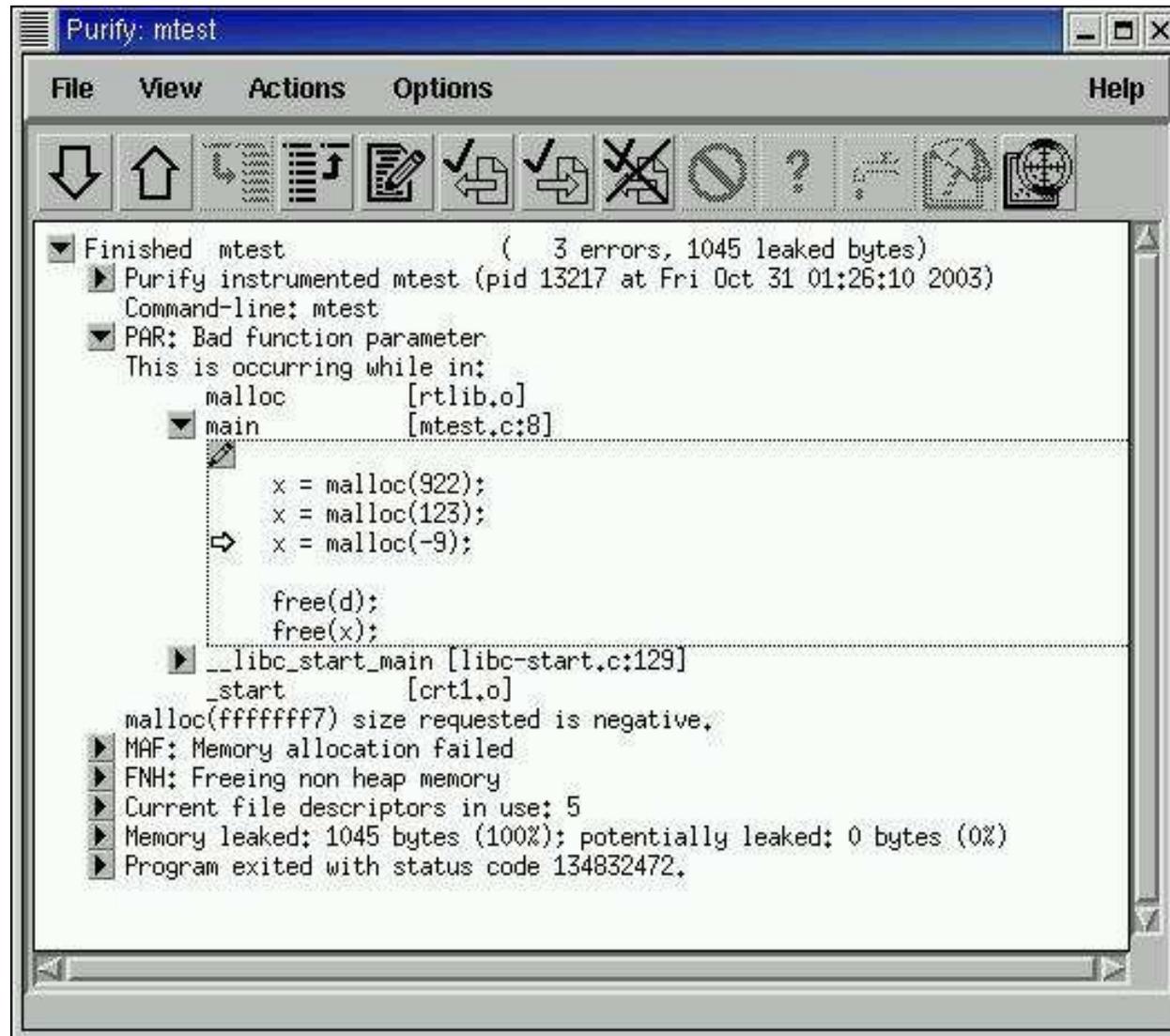
Valgrind

- <http://developer.kde.org/~sewardj/>
 - Last stable version 20031012
 - Only on x86-Linux
 - Works on many distributions, but not all
 - Yes: RH 7.2 7.3 8 9
 - No: RH 7.1
 - `kcachegrind` GUI only available under KDE
 - `memcheck` slows down execution by 25-50 times
 - `addrcheck` is lighter weight, but does not track read-before-write's
 - the `-gen-suppressions=yes` option tells Valgrind to print out a suppression for each error that appears
-

IBM Rational PurifyPlus

- <http://www.rational.com/>
 - Runtime analysis
 - Memory corruption detection
 - Memory leakage detection
 - Requires instrumentation
 - `purify gcc -g mtest.c`
 - Languages: C, C++
-

Purify: Bad Function Parameter



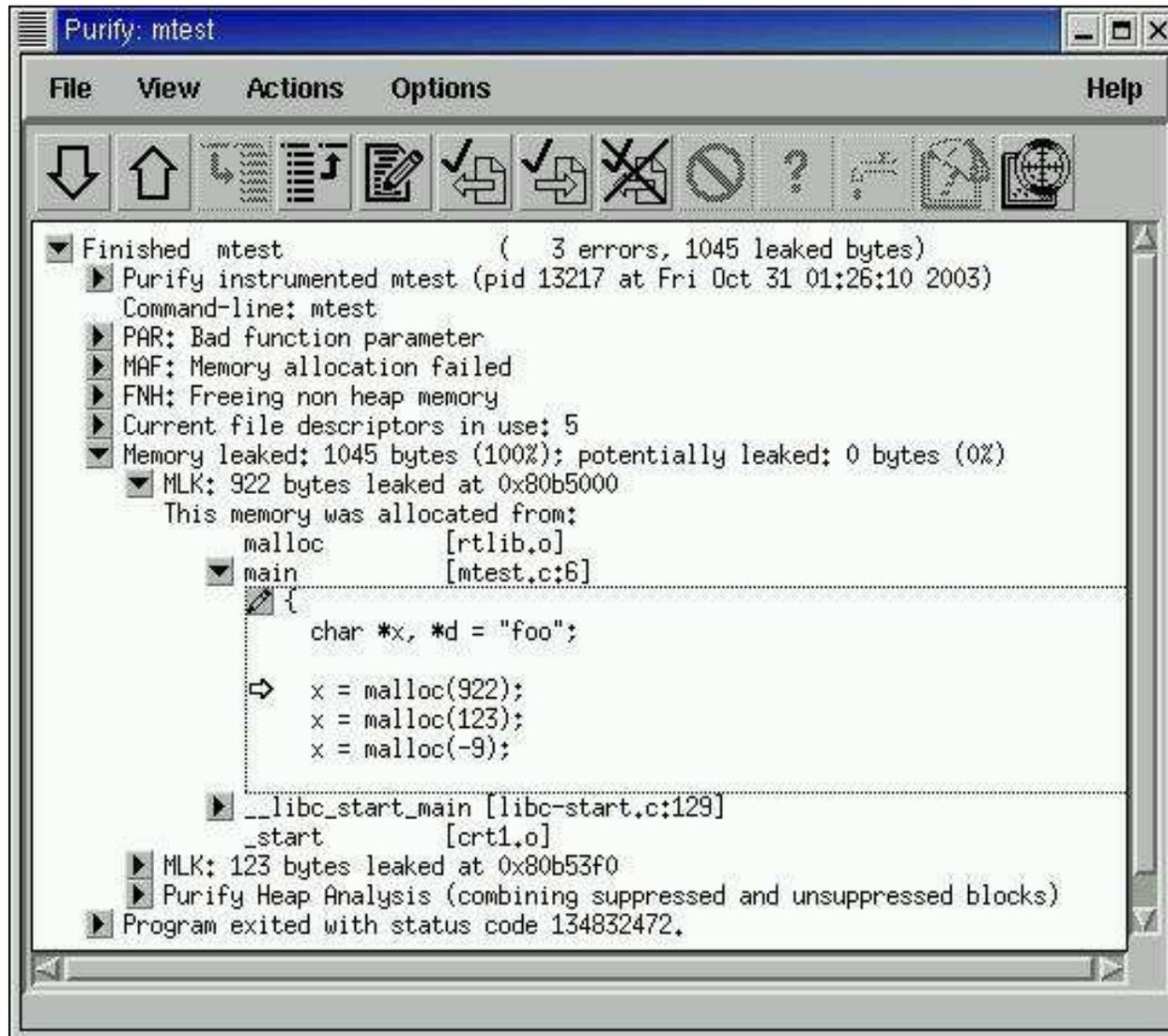
The screenshot shows the Purify: mtest application window. The title bar reads "Purify: mtest". The menu bar includes "File", "View", "Actions", "Options", and "Help". Below the menu bar is a toolbar with various icons. The main content area displays a stack trace for a "Bad function parameter" error. The error message is "PAR: Bad function parameter" and it indicates that this is occurring while in "main" at "mtest.c:8". The stack trace shows the following frames:

- Finished mtest (3 errors, 1045 leaked bytes)
 - Purify instrumented mtest (pid 13217 at Fri Oct 31 01:26:10 2003)
 - Command-line: mtest
 - PAR: Bad function parameter
 - This is occurring while in:
 - malloc [rtlib.o]
 - main [mtest.c:8]
 - x = malloc(922);
 - x = malloc(123);
 - x = malloc(-9);
 - free(d);
 - free(x);
 - __libc_start_main [libc-start.c:129]
 - _start [crt1.o]

Additional error messages at the bottom of the window include:

- malloc(ffffffff7) size requested is negative.
- MAF: Memory allocation failed
- FNH: Freeing non heap memory
- Current file descriptors in use: 5
- Memory leaked: 1045 bytes (100%); potentially leaked: 0 bytes (0%)
- Program exited with status code 134832472.

Purify: Memory Leaks

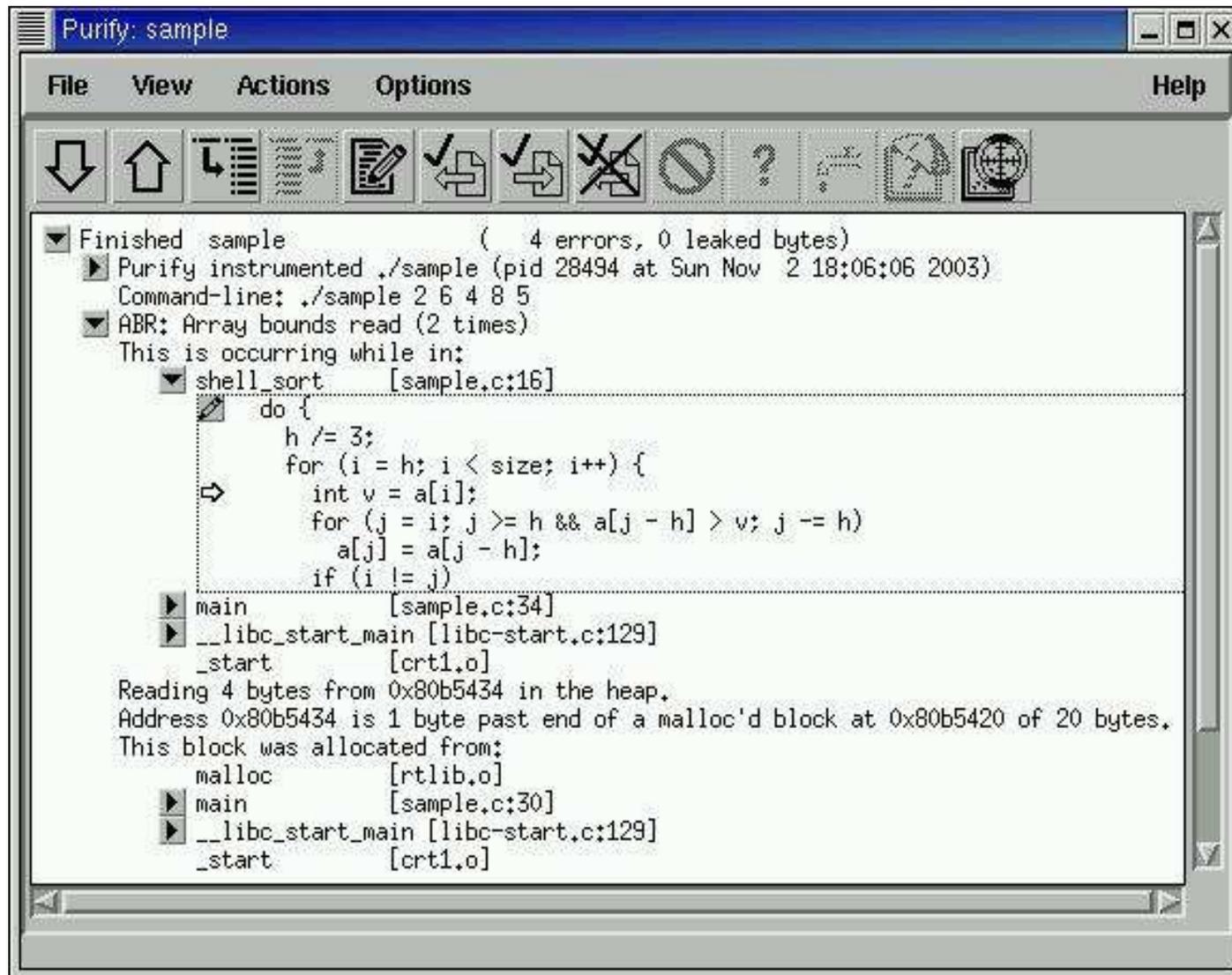


A Sample Program

```
int main(int argc, char *argv[])
{
    int *a, i;

    a = (int *)malloc((argc - 1) * sizeof(int));
    for (i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);
    shell_sort(a, argc);
    for (i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

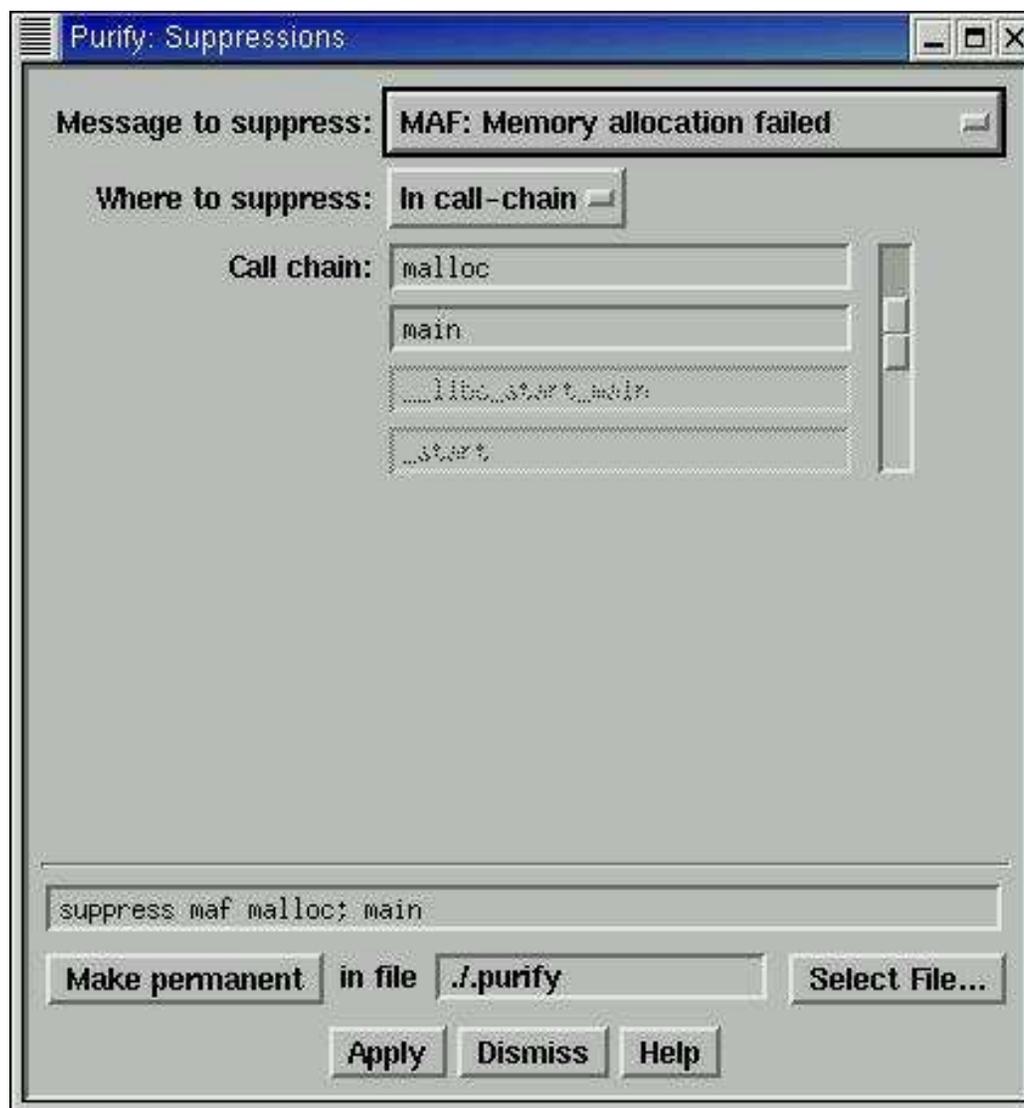
Purify: Out-of-Bounds Read



The screenshot shows the Purify application window titled "Purify: sample". The window has a menu bar with "File", "View", "Actions", "Options", and "Help". Below the menu bar is a toolbar with various icons. The main content area displays the following information:

```
Finished sample ( 4 errors, 0 leaked bytes)
  Purify instrumented ./sample (pid 28494 at Sun Nov  2 18:06:06 2003)
  Command-line: ./sample 2 6 4 8 5
  ABR: Array bounds read (2 times)
  This is occurring while in:
    shell_sort [sample.c:16]
      do {
        h /= 3;
        for (i = h; i < size; i++) {
          int v = a[i];
          for (j = i; j >= h && a[j - h] > v; j -= h)
            a[j] = a[j - h];
          if (i != j)
            continue;
        }
      }
    main [sample.c:34]
    __libc_start_main [libc-start.c:129]
    _start [crt1.o]
  Reading 4 bytes from 0x80b5434 in the heap.
  Address 0x80b5434 is 1 byte past end of a malloc'd block at 0x80b5420 of 20 bytes.
  This block was allocated from:
    malloc [rtlib.o]
    main [sample.c:30]
    __libc_start_main [libc-start.c:129]
    _start [crt1.o]
```

Purify: Suppressions



Purify: Library Functions

- Library functions allow developer to customize data collected for a given application
- Memory usage profiling:

```
#ifndef PURIFY
...
purify_all_inuse();
...
#endif
```

Used in VIS together with a couple of scripts to profile memory usage on a per-package basis

- Link to `libpurify_stubs.a`
-

The Cost of Instrumentation

- One data point

no instrumentation	64 s
valgrind -skin=addrcheck	860 s
valgrind -skin=memcheck	1287 s
purify	1725 s

- The `addrcheck` skin checks the validity of addresses but not of data
- Only `purify` detects this uninitialized memory read

```
int main()  
{  
    int a;  
    return a;  
}
```

GDB and IDB

- Better used through a graphical front-end
 - Ddd
 - emacs's GUD
 - UPS (<http://ups.sourceforge.net/main.html>)
 - Insight (<http://sources.redhat.com/insight/>)
 - GDB and IDB largely compatible
 - `idb -gdb` is similar to `gdb`
 - otherwise, it is similar to `dbx`
 - Both can be used with the “other” compiler
 - There are other debuggers as well
 - TotalView
 - Idebug (Java)
-

The Data Display Debugger

- Front-end for
 - C/C++ (gdb, idb)
 - Other languages supported by gcc (e.g., Fortran)
 - Perl
 - Python
 - Java
- Available also for other operating systems
 - Works with other inferior debuggers too (e.g., dbx)
 - Requires X server
- <http://www.gnu.org/software/ddd>



The Data Display Debugger

The screenshot shows the Data Display Debugger (DDD) interface with the following components labeled:

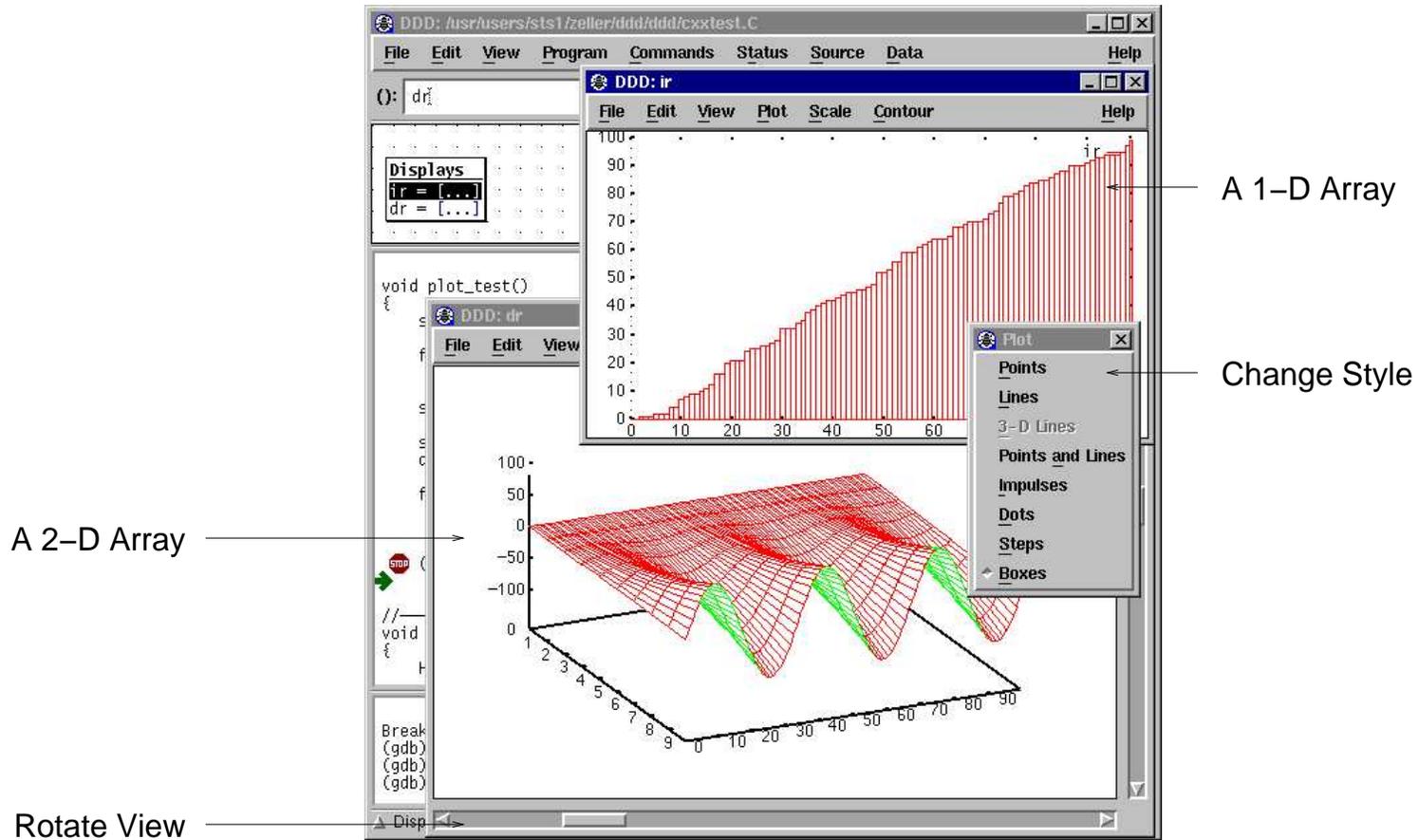
- Menu Bar:** File, Edit, View, Program, Commands, Status, Source, Data, Help
- Tool Bar:** Lookup, Find, Break, Watch, Print, Display, Hide, Rotate, Set, Undisp
- Data Window:** Displays memory locations and their values. For example, `1: list (List *) 0x804ab78` with `value = 85`, `self = 0x804ab78`, and `next = 0x804ab88`.
- Source Window:** Shows the source code for `list_test`. A red stop icon indicates the current execution point.
- Machine Code Window:** Shows the assembly code for the current instruction, such as `0x8048a27 <list_test__Fi+151>: movl 0xffffffff(%ebp),%eax`.
- Debugger Console:** Shows GDB commands and their output, such as `(gdb) graph display *list dependent on 1`.
- Status Line:** Shows the current register values, such as `$edx = 0x804ab78 (134523768)`.
- Command Tool:** A vertical toolbar with buttons for Run, Interrupt, Step, Next, Until, Cont, Up, Back, Edit, Step1, Next1, Finish, Kill, Down, Fwd, and Make.
- Value Tip:** A tooltip showing the value `0x804ab78 (134523768)`.
- Busy Indicator:** A small icon in the bottom right corner.

The DDD Layout using Stacked Windows

DDD: Displaying Data

- (gdb) `graph display array[0] @ nelem`
 - Shows **array slice** in the data window
 - Optionally use `rotate` button for more compact display
 - (gdb) `graph plot array[0] @ nelem`
 - runs `gnuplot` on array slice and displays result in new window
 - Plot is updated when data changes
 - Plot can be customized and saved
 - Animations are possible
-

DDD: Plotting



Plotting 1-D and 2-D Arrays

DDD: Machine-Level Debugging

The register name
is copied to ()

Select register

The screenshot shows the DDD debugger interface. The main window displays a memory dump of assembly code. A secondary window titled "DDD: Registers" is open, showing a list of registers and their values. The register "esp" is selected and highlighted. The "Registers" window has a table with the following data:

Register	Value 1	Value 2
eax	0x2	2
ecx	0xb	11
edx	0x1d	29
ebx	0x0	0
esp	0xbffff958	0xbffff958
ebp	0xbffff964	0xbffff964
esi	0x4000623c	1073766972
edi	0x80487c0	134514624
eip	0x8048955	0x8048955
eflags	0x296	662
cs	0x23	35
ss	0x2b	43

Below the table, there are radio buttons for "Integer registers" and "All registers". The "Integer registers" option is selected. At the bottom of the window are "Close" and "Help" buttons.

The main window shows a memory dump with addresses from 0x8048934 to 0x8048973. The instruction at 0x8048946 is "int, int, int)". A red stop sign icon is visible next to the instruction at 0x8048955. The status bar at the bottom left shows "\$eax = 0x2 (2)".

Displaying Register Values

Profiling

- Gcov
 - Gprof
 - VTune
 - Valgrind
-

Optimization Tips

- Static branch prediction in the Pentium 4
 - Forward branches are not taken
 - Backward branches are taken
 - Use `const`; avoid `register`
 - Fit data structures to cache lines
 - More at <http://developer.intel.com/design/pentium4/manuals/>
 - Profiling tools help identify
 - hotspots
 - inefficient memory layout
 - insufficiently tested code
 - Remember: Only optimize what is critical
-

Sampling vs. Counting

- Sampling: the program counter is periodically examined
 - Basic block counting: the executable is instrumented so that the frequencies of execution of all basic blocks are recorded
 - Only reliable mechanism for
 - coverage measurement
 - fine tuning
 - Does not account for memory hierarchy
-

Gcov: Coverage Analysis

- `gcc -fprofile-arcs -ftest-coverage -o lfsr lfsr.c`

- `./lfsr`

- `gcov lfsr.c`

```
100.00% of 10 source lines executed in file lfsr.c  
Creating lfsr.c.gcov.
```

Gcov: Coverage Analysis

```
        int main()  
1      {  
1      unsigned int r = 1;  
1      int i;  
10000000001  for (i = 0; i < 10000000000; i++) {  
10000000000      unsigned int b = r & 1;  
10000000000      r >>= 1;  
10000000000      if (b)  
500007631          r ^= 0x8805;  
        }  
1      printf("%u\n", r);  
1      exit(0);  
    }
```

Gprof

```
gcc -o lfsr -g -pg -fprofile-arcs -O3 \  
  -mcpu=pentium4 lfsr.c  
./lfsr  
gprof --line --flat-profile lfsr
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	name
61.59	8.11	8.11	main (lfsr.c:13)
28.91	11.91	3.81	main (lfsr.c:17)
3.19	12.33	0.42	main (lfsr.c:14)
3.19	12.75	0.42	main (lfsr.c:15)
2.89	13.13	0.38	main (lfsr.c:16)
0.23	13.16	0.03	main (lfsr.c:14)

IBM Rational Quantify and Purecov

- Basic-block counting profiling
- Call graph analysis
- Source annotation

Intel VTune for Linux

- vt1: command line version of the performance analyzer for Linux
 - Sampling: non-intrusive, system-wide profiling
 - relies on the CPU performance monitoring registers
 - Call graph: low overhead analysis of program flow
 - requires instrumentation
 - <http://www.intel.com/software/products/vtune/vlin/>
 - Current release is 1.1
 - Several Red Hat and SUSE releases supported
-

VTune: Sampling

- `vtl activity -c sampling run`
 - Runs the sampling collector for all processes
 - Automatically calibrates collection parameters
 - Collects data on clock ticks and retired instructions
 - `vtl show`
 - Displays activities that have been run for a [project](#)
 - `vtl view a1::r1 -processes`
 - Presents the results of activity `a1::r1` organized by process
 - `vtl -help -c sampling`
 - Shows what events can be sampled
-

VTune: Call Graph

- `vtl activity -c callgraph -app ./mypgm \`
`-moi ./mypgm run`
 - Runs the callgraph collector for `mypgm`
 - Performs instrumentation (including library functions)
 - Collects function call data
 - `vtl show`
 - Displays activities that have been run for a [project](#)
 - `vtl view a1::r1 -functions`
 - Shows timing information for each function
 - Use `-calls` for call-graph edge data
 - `vtl view a1::r1 -critical-path`
 - Shows the critical path
-

Valgrind: Cache Profiling

- Valgrind contains built-in support for cache profiling
 - `valgrind -skin=cachegrind my-program`
 - detailed simulation of L1-D, L1-I, unified L2
- `vg_annotate` annotates source code
- Cache configuration auto-detected using the CPUID instruction
 - can be overridden

Valgrind: Cache Profiling

```
I  refs:          73,173,467
I1 misses:         70,260
L2i misses:         1,734
I1 miss rate:      0.9%
L2i miss rate:     0.0%
```

```
D  refs:          39,315,546 (28,535,016 rd + 10,780,530 wr)
D1 misses:         456,530 ( 344,528 rd + 112,002 wr)
L2d misses:         249,456 ( 162,814 rd + 86,642 wr)
D1 miss rate:      1.1% ( 1.2% + 1.0% )
L2d miss rate:     0.6% ( 0.5% + 0.8% )
```

```
L2 refs:           526,790 ( 414,788 rd + 112,002 wr)
L2 misses:         251,190 ( 164,548 rd + 86,642 wr)
L2 miss rate:      0.2% ( 0.1% + 0.8% )
```

The End

