Automatically Mining Program Build Information via Signature Matching

Charng-Da Lu, Matt Jones, Tom Furlani
Center for Computational Research
SUNY at Buffalo

July 2011
More than 10 years experience delivering HPC in an academic setting

~ 1K jobs per day, 720K CPU days delivered (2009)

Hardware
- 1024 3.0/3.2GHz Xeon cores, 512 nodes, Myrinet MX
- 1024 2.27 GHz Nehalem cores, 128 nodes, IB/Mellanox
- 1024 2.13 GHz Westmere cores, 128 nodes, IB/Mellanox
- 384 2.67 GHz Westmere cores, 32 nodes, IB/Mellanox, nVidia Tesla M2050 GPUs
- ...

Software
- Intel, PGI, GCC, nVidia NVCC compilers.
- Intel MPI, Open MPI, MVAPICH, MPICH-MX...
Software maintained at our center
Software maintained on Kraken

(Credit: ALTD by B. Hadri, M. Fahey and N. Jones)
Motivation

To understand and monitor software and library usage on HPC systems

- Interest from Funding Agencies
  - To gauge how HPC resources meet their funding initiatives and scientific goals

- TG Technology Audit Service
  - To relate performance changes with program builds

- HPC Centers
  - To reduce costs and management overhead by retiring rarely used software
ALTD

*Automatic Library Tracking Database*

- TG ’10 paper by B. Hadri, M. Fahey and N. Jones from Oak Ridge
- Automatically tracks library usage at compilation and at execution times

*But..*

- Only works for Cray
  - Static linking, wrappers for linker & job launcher
- Only works after ALTD is installed
Our Approach

- Analyze program binaries by scanning for known “signatures” (hex strings)
- Similar to how anti-virus software detects malware.
- Signature sources:
  - Compiler-specific code snippets.
  - Compiler-specific meta data.
  - Library code snippets.
  - Symbol versioning.
  - Checksums.
Compiler-Specific Code Snippets

- Example: “Processor Dispatch” by certain optimizing compilers (Intel, PGI)
- To handle the discrepancy between the processors the program built for and the processors the program actually runs on.
- x86 instruction set evolution: SSE
Compiler-Specific Code Snippets (Cont.)

<table>
<thead>
<tr>
<th>SSE Generation</th>
<th>Intel Processors</th>
<th>AMD Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVX</td>
<td>Sandy Bridge</td>
<td>Bulldozer</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>Westmere</td>
<td></td>
</tr>
<tr>
<td>SSE4.1</td>
<td>Nehalem</td>
<td></td>
</tr>
<tr>
<td>SSE4a</td>
<td></td>
<td>Barcelona</td>
</tr>
<tr>
<td>Supplemental SSE3</td>
<td>Merom (Core)</td>
<td></td>
</tr>
<tr>
<td>SSE3</td>
<td>Prescott (Pentium 4)</td>
<td>K8</td>
</tr>
</tbody>
</table>

° Try to run a program tuned for AVX on non-AVX-capable processors (built with Intel compiler):

"Fatal Error: This program was not built to run on the processor in your system.

The allowed processors are: Intel(R) processors with Intel(R) AVX instructions support."
char *memcpy(...) {
    switch (SSE Capability) {
    case SSE3: Run_SSE3_memcpy(...);
    case SSE4: Run_SSE4_memcpy(...);
    case AVX:  Run_AVX_memcpy(...);
    default:   Run_SSE2_memcpy(...);
    }
}
Compiler-Specific Meta Data

Linux Program Binary File Structure:
- File Header
- .text
- .data
- .rodata
- .bss
- Symbol Tables
- Relocation Tables
- (other sections)
- Symbol Version Tables
- String Tables

.Symbol Tables
- Relocation Tables
- (other sections)
- Symbol Version Tables
- String Tables

..comment section contains compiler brand strings:
- GCC: (GNU) 4.1.2 20080704 (Red Hat 4.1.2-50)
- PGF90 11.0-0
- PathScale Compiler Version 4.0.5: hello.c compiled with -O2 -march=wo ...
- -defaultlib:libirc ...
- GCC: (GNU) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2.8)
- CRAY C: 7.2.3 (u7262c72173i721 ...
- acomp: Sun C 5.11 Linux_i386 2010/08/13

- Use “readelf –p .comment a.out” command to see them.
- The “strip” utility does **NOT** remove .comment
Static Library Code Snippets (Cont.)

Static linking examples

Compiler-specific code snippets
- Intel: libirc.a
- PGI: libpgc.a
- GCC: crt*.o, libgcc.a

Cray XT programming environment

MPI libraries such as MPICH, MVAPICH

IO libraries such as HDF, NetCDF

Some Fortran runtime libraries
- Intel: libifcore.a
- PGI: libpgf90.a
- GFortran: libgfortran.a
Symbol Versioning

- Some Dynamic Linking Libraries can be identified by their Symbol Version tables

**Symbol Versioning:** Introduced by Sun in 1995 to manage DLL versions.

- Each symbol (subroutine name) in a DLL is associated with a version (a string).
- Versions form a chain.
- The prefix in a Version string indicates the library.
- The highest version number is the version of the library.

- Used extensively by GNU development tools, Infiniband/Open Fabrics, Myrinet MX, DAPL...

```bash
libc {
    GLIBC_2.0 {
        malloc;
        free;
    }
    GLIBC_2.1 {
        malloc_info;
    }
    GLIBC_2.10 {
        malloc_info;
    }
    ...
}
```

Version definition script of Glibc
Recap

- To identify the compiler
  - Compiler-specific code snippets
  - Compiler-specific meta data
- To identify the static libraries
  - Library code snippets
- To identify the dynamic libraries
  - Symbol versioning
  - MD5 checksums
Implementation

° Based on the anti-virus package ClamAV
  ▶ Production Quality, Open Source
  ▶ High Performance Scanning Engine
    • Wu-Manber algorithm
    • Aho-Corasick algorithm
  ▶ Signature Expressiveness

° ClamAV Signature Formats
  ▶ Basic: Plain hex-strings, e.g. 48f7e748894424404889f0488 ...
  ▶ Regular Expression:
    • ?? (one byte)
    • ?a (high nibble)
    • * (any bytes)
    • (aa|bb|..) (match aa or bb or ..)
    • {n} (n bytes)
    • {n-m} (btw n and m bytes)
    • {-n} (n or less bytes)
    • {n-} (n or more bytes)

▶ MD5 Checksum
Signature Generation & Matching

Library #1
- File Header
- .text
- .data
- .rodata
- (other sections)

Library #n
- File Header
- .text
- .data
- .rodata
- (other sections)

Program Binary
- File Header
- .text
- .data
- .rodata
- (other sections)

Signature Generator

Signature Metadata

ClamAV Signatures

Signature Library
- (58 times, 346766 bytes) PGI Fortran Compiler 11.x
- (48 times, 56833 bytes) PGI Fortran Compiler 8.x
- (45 times, 118288 bytes) PGI Fortran Compiler 10.x
- (42 times, 49895 bytes) PGI Fortran Compiler 7.x
- (32 times, 82808 bytes) PGI Compiler Suite 11.x
- (29 times, 57166 bytes) PGI Compiler Suite 7.x
- (2 times, 200 bytes) GCC 4.4.3

User Annotation
- (Name, Vendor, Version, ...)

Signature Database

Signature Scanner
Signature Generator

1. Parse object and library files
2. Extract machine code from .text sections
3. Apply the relocations
4. Trim long signatures

```c
void foo() { 
    char *buf = malloc(10);
}
```

```
000000 <foo>:
  0: 55              push   %rbp
  1: 48 89 e5        mov    %rsp,%rbp
  4: 48 83 ec 10     sub    $0x10,%rsp
  8: bf 0a 00 00 00  mov    $0xa,%edi
 d: e8 00 00 00 00  callq  12 <foo+0x12>
12: 48 89 45 f8     mov    %rax,-0x8(%rbp)
16: c9              leaveq
17: c3              retq
```

554889e54883ec10bf0a0000 00e8{4}488945f8c9c3

ClamAV Regex signature

Compiler

Extract

Trim

Relocate

Signature Scanner
## Compiler Identification Tests

### Fourteen x86 64-bit compilers examined

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Code Snippet Source</th>
<th>Compiler Metadata</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absoft</td>
<td>11.1</td>
<td>liba*.a</td>
<td></td>
<td>Fortran only</td>
</tr>
<tr>
<td>Clang</td>
<td>2.8</td>
<td></td>
<td></td>
<td>C/C++ only</td>
</tr>
<tr>
<td>Cray</td>
<td>7.1, 7.2</td>
<td>libcsup.a, libf*.a, libcray*.a</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>G95</td>
<td>0.93</td>
<td>libf95.a</td>
<td>V</td>
<td>Fortran only</td>
</tr>
<tr>
<td>GNU</td>
<td>4.1, 4.4, 4.5</td>
<td>crt*.o, libgcc*.a</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>Intel</td>
<td>9 thru 12</td>
<td>libirc*.a, libfcore*.a</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Lahey-Fujitsu</td>
<td>8.1</td>
<td>fj*.o, libfj*.a</td>
<td>I</td>
<td>Fortran only</td>
</tr>
<tr>
<td>LLVM-GCC</td>
<td>2.8</td>
<td></td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>NAG</td>
<td>5.2</td>
<td>libf*.a</td>
<td></td>
<td>Fortran only</td>
</tr>
<tr>
<td>Open64</td>
<td>4.2.x</td>
<td>libopen64*.a, libf*.a</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PathScale</td>
<td>3.2, 4.0</td>
<td>lib*.crt.a, libpath*.a</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PCC</td>
<td>0.99</td>
<td>crt*.o, libpcc*.a</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>PGI</td>
<td>6 thru 11</td>
<td>libpgc.a, libpgf*.a, f90*.o, pgf*.o</td>
<td>V</td>
<td>C/C++ only</td>
</tr>
<tr>
<td>Sun Studio</td>
<td>12.x</td>
<td>crt*.o, libc_supp.a, libf*.a</td>
<td>V</td>
<td></td>
</tr>
</tbody>
</table>

V: Can identify both compiler & its version  
I: Can identify compiler only
Compiler Identification Tests

- **Typical Output**
  - (58 times, 346766 bytes) PGI Fortran Compiler 11.x
  - (48 times, 56833 bytes) PGI Fortran Compiler 8.x
  - (45 times, 118288 bytes) PGI Fortran Compiler 10.x
  - (42 times, 49895 bytes) PGI Fortran Compiler 7.x
  - (32 times, 82808 bytes) PGI Compiler Suite 11.x
  - (29 times, 57166 bytes) PGI Compiler Suite 7.x
  - (2 times, 200 bytes) GCC 4.4.3
  - (3 times, 6992 bytes) Intel Compiler Suite 12.0
  - (2 times, 200 bytes) GCC 4.4.3

- **Observations**
  - GCC is the common denominator
    - Compilers use GNU-specific code snippets to achieve compatibility with GNU dev tools.
  - Fortran compilers can be identified by their runtime libraries
    - Not possible for C/C++ as Glibc is the standard.
  - Some compilers reuse code across releases
    - Multiple versions matches are possible.
  - Some compilers share a common codebase
    - Only compiler-specific metadata can differentiate them (e.g. Open64 v. PathScale)
## Library Tests

### Test signature database

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>Code Snippet Source</th>
<th>Mean and StdDev</th>
<th>Built with/for which compiler(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACML (AMD Core Math Lib)</td>
<td>4.4.0</td>
<td>libacml*.a</td>
<td>11.1, 7.8</td>
<td>Intel, PGI</td>
</tr>
<tr>
<td>Cray LibSci</td>
<td>10.4.0</td>
<td>libsci*.a</td>
<td>3.4, 4.9</td>
<td>Intel, PGI, GNU</td>
</tr>
<tr>
<td>Intel MKL (Math Kernel Lib)</td>
<td>8.0, 9.0</td>
<td>libmkl*.a</td>
<td>4.6, 9.0</td>
<td>Intel</td>
</tr>
<tr>
<td></td>
<td>10.x</td>
<td>libmkl_core.a</td>
<td>4.2, 16.6</td>
<td></td>
</tr>
<tr>
<td>Cray MPI</td>
<td>3.5.1</td>
<td>libmpich*.a</td>
<td>1.3, 2.6</td>
<td>Intel, PGI, GNU</td>
</tr>
<tr>
<td>MPICH</td>
<td>1.2.7mx</td>
<td>libmpich.a</td>
<td>1.2, 2.7</td>
<td>Intel, GNU</td>
</tr>
<tr>
<td>MVAPICH</td>
<td>1.4, 1.5</td>
<td>libmpich.a</td>
<td>2.6, 4.8</td>
<td>Intel</td>
</tr>
</tbody>
</table>

- **100K signatures in total.**
- **It’s fast to generate signatures**
  - E.g. Intel MKL 10.3.1’s libmkl_core.a
    - 210 MB in size
    - ~44,000 object files in it
    - Largest object file: 1.4 MB
    - 28 seconds to complete on 2.8 GHz Xeon (single-threaded)
    - 78% time spent in I/O
21 test program binaries

- From real-world HPC software packages
  - Amber, Charmm, CPMD, GAMESS, Lammps, NAMD, NWChem, Quantum ESPRESSO
  - Built on a generic Intel PC cluster and a Cray XT5m
- Mean .text size: **13.4 MB**
- Largest code: NWChem 6.0 on Cray XT5m, **39.4 MB**

Signature scanning performance

- **2.5 GHz Intel Xeon L5420 Harpertown**
- **2.8 GHz Intel Xeon X5560 Nehalem**
## Library Identification

### Sample Output

- **Amber 11’s pmemd (Cray XT5m)**
  - **COMPILER:** PGI Fortran Compiler 11.x (66 times, 363088 bytes)
  - **COMPILER:** PGI Fortran Compiler 8.x (51 times, 60225 bytes)
  - **COMPILER:** PGI Fortran Compiler 10.x (48 times, 121680 bytes)
  - **COMPILER:** PGI Fortran Compiler 7.x (46 times, 53297 bytes)
  - **COMPILER:** PGI Compiler Suite 10.x (36 times, 60410 bytes)
  - **COMPILER:** PGI Compiler Suite 7.x (36 times, 58744 bytes)
  - **COMPILER:** PGI Compiler Suite 11.x (36 times, 84054 bytes)
  - **COMPILER:** PGI Compiler Suite 8.x (32 times, 54939 bytes)
  - **COMPILER:** PGI Fortran Compiler 6.x (13 times, 2672 bytes)
  - **COMPILER:** PGI Compiler Suite 6.x (11 times, 2466 bytes)
  - **COMPILER:** GCC 4.5.0 (2 times, 248 bytes)
  - **STATICLIB:** Cray MPI 3.5.1 (PGI, Intel Compiler, GCC) (188 times, 429214 bytes)

- **NWChem 5.1.1 (Intel PC cluster)**
  - **COMPILER:** Intel Fortran Compiler 11.1 (77 times, 661504 bytes)
  - **COMPILER:** Intel Compiler Suite 11.1 (18 times, 60096 bytes)
  - **COMPILER:** Intel Fortran Compiler 12.0 (4 times, 128 bytes)
  - **COMPILER:** Intel Fortran Compiler 11.0 (3 times, 96 bytes)
  - **COMPILER:** GCC 4.1.2 (2 times, 188 bytes)
  - **DYNLIB:** Intel MKL 10.2.2
  - **DYNLIB:** Glibc 2.5
  - **DYNLIB:** GCC Runtime Support Library 4.2.0
  - **DYNLIB:** InfiniBand Verbs Library 1.1
  - **STATICLIB:** MVAPICH2 1.4.x (Intel Compiler 11.1) (171 times, 624768 bytes)
  - **STATICLIB:** MVAPICH2 1.5.x (Intel Compiler 11.1) (16 times, 29424 bytes)
**Integration with TAS**

- Integral part of TG Technology Audit Service
  - Automatic discovery and documentation of program builds.
  - Performance correlation with program changes.
  - Example: MPI Tile IO
    - 3D array collective write
    - 64 processes
    - Panasas filesystem

---

**Throughput (MB/s)**

Intel MPI 4.0

Intel MPI 4.0.1 supports Panasas MPI-IO hint `panfs_concurrent_write`
Future Work

° Automatic Library Tracking Database for generic PC clusters
  ☑ Modify “module load..” scripts
  ☑ Modify “mpicc” compiler wrappers
  ☑ Integrate with PBS job scheduler
  ☑ Scan users’ program binaries
Caveat

° Our approach is not guaranteed to work if
  - Compiler optimization is turned off explicitly \((-O0)\) or implicitly \((-g)\).
  - Multiple compilers are used.
  - Non-x86 platforms.

° Our approach can’t discover compiler flags used
  - \(-\text{sox} \ (\text{Intel}), -\text{frecord-gcc-switches} \ (\text{GNU})\) can record the flags.
  - PathScale/Open64/Absoft record the flags automatically.
Conclusion

° Signature-based approach to automatic discovery of compiler and library information of program binaries.
° Signature generation is simple and straightforward; no expert knowledge is required.
° Signature scanning is based on a production quality, high performance scanning engine.
° Already in production in TG Technology Audit Service.
Q & A

° ???????