Proceedings of the
Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

The 7 dwarves: debugging information beyond gdb

Arnaldo Carvalho de Melo
Red Hat, Inc.
acme@redhat.com
acme@ghostprotocols.net

Abstract

The DWARF debugging information format has been so far used in debuggers such as gdb, and more recently in tools such as systemtap and frysk.

In this paper the author will show additional scenarios where such information can be useful, such as:

- Showing the layout of data structures;
- Reorganizing such data structures to remove alignment holes;
- Improving CPU cache utilization;
- Displaying statistics about inlining of functions;
- Re-creating structs and functions from the debugging information;
- Showing binary diffs to help understand the effects of any code change.

And much more.

1 Introduction

This paper talks about new ways to use the DWARF debugging information inserted into binaries by compilers such as gcc.

The author developed several tools that allow:

- Extracting useful information about data structures layout;
- Finding holes and padding inserted by the compiler to follow alignment constraints in processor architectures;
- To find out possibilities for reduction of such data structures;
- Use of information about function parameters and return types to generate Linux kernel modules for obtaining data needed for generation of callgraphs and values set to fields at runtime;
- A tool that given two object files shows a binary diff to help understanding the effects of source code changes on size of functions and data structures.

Some use cases will be presented, showing how the tools can be used to solve real world problems.

Ideas discussed with some fellow developers but not yet tried will also be presented, with the intent of *hopefully* having them finally tested in practice by interested readers.

2 DWARF Debugging Format

DWARF [3] is a debugging file format used by many compilers to store information about data structures, variables, functions and other language aspects needed by high level debuggers.

It has had three major revisions, with the second being incompatible with the first, the third is an expansion of the second, adding support for more C++ concepts, providing ways to eliminate data duplication, support for debugging data in shared libraries and in files larger than 4 GB.

The DWARF debugging information is organized in several ELF sections in object files, some of which will be mentioned here. Please refer to the DWARF [3] specification for a complete list. Recent developments in tools such as elfutils [2] allow for separate files with the debugging information, but the common case is for the information to be packaged together with the respective object file.

Debugging data is organized in tags with attributes. Tags can be nested to represent, for instance, variables inside lexical blocks, parameters for a function and other hierarchical concepts.

As an example let us look at how the ubiquitous hello world example is represented in the ".debug_info" section, the one section that is most relevant to the subject of this paper:

```
$ cat hello.c
int main(void)
{
        printf("hello, world!\n");
}
```

Using gcc with the -g flag to insert the debugging information:

```
$ gcc —g hello.c —o hello
```

Now let us see the output, slightly edited for brevity, from eu-readelf, a tool present in the elfutils package:

```
$ eu—readelf —winfo hello
DWARF section ,.debug_info, at offset
0x6b3:
[Offset]
Compilation unit at offset 0:
Version: 2, Abbrev section offset: 0,
Addr size: 4, Offset size: 4
[b] compile_unit
     stmt_list   0
     high_pc     0x0804837a
     low_pc      0x08048354
     producer    "GNU C 4.1.1"
     language    ISO C89 (1)
     name        "hello.c"
     comp_dir    "~/examples"
[68] subprogram
      external
      name       "main"
      decl_file  1
      decl_line  2
      prototyped
      type       [82]
      low_pc     0x08048354
      high_pc    0x0804837a
      frame_base location list [0]
[82] base_type
      name       "int"
      byte_size  4
      encoding   signed (5)
```

Entries starting with [*number*] are the DWARF tags that are represented in the tool's source code as `DW_TAG_tag_name`. In the above output we can see some: `DW_TAG_compile_unit`, with information about the object file being analyzed, `DW_TAG_subprogram`, emitted for each function, such as "main" and `DW_TAG_base_type`, emitted for the language basic types, such as `int`.

Each tag has a number of attributes, represented in source code as `DW_AT_attribute_name`. In the `DW_TAG_subprogram` for the "main" function we have some: `DW_AT_name` ("main"), `DW_AT_decl_file`, that is an index into another DWARF section with the names for the source code files, `DW_AT_decl_line`, the line in the source code where this function was defined, `DW_AT_type`, the return type for the "main" routine. Its value is a tag index, that in this case refers to the [82] tag, which is the `DW_TAG_base_type` for "int," and also the address of this function, `DW_AT_low_pc`.

The following example exposes some additional DWARF tags and attributes used in the seven dwarves. The following struct:

```
struct swiss_cheese {
  char a;
  int  b;
};
```

is represented as:

```
[68] structure_type
      name          "swiss_cheese"
      byte_size     8
[7d]  member
       name         "a"
       type         [96]
       data_member_location 0
[89]  member
       name         "b"
       type         [9e]
       data_member_location 4
[96] base_type
       name         "char"
       byte_size    1
```

```
[9e] base_type
       name          "int"
       byte_size     4
```

In addition to the tags already described we have now `DW_TAG_structure_type` to start the representation of a struct, that has the `DW_AT_byte_size` attribute stating how many bytes the struct takes (8 bytes in this case). There is also another tag, `DW_TAG_member`, that represents each struct member. It has the `DW_AT_byte_size` and the `DW_AT_data_member_location` attribute, the offset of this member in the struct. There are more attributes, but for brevity and for the purposes of this paper, the above are enough to describe.

3 The 7 dwarves

The seven dwarves are tools that use the DWARF debugging information to examine data struct layout (`pahole`), examine executable code characteristics (`pfunct`), compare executables (`codiff`), trace execution of functions associated with a struct (`ctracer`), pretty-print DWARF information (`pdwtags`), list global symbols (`pglobal`), and count the number of times each set of tags is used (`prefcnt`).

Some are very simple and still require work, while others, such as `pahole` and `pfunct`, are already being helpful in open source projects such as the Linux kernel, xine-lib and perfmon2. One possible use is to pretty-print DWARF information accidentally left in binary-only kernel modules released publicly.

All of these tools use a library called `libdwarves`, that is packaged with the tools and uses the DWARF libraries found in elfutils [2]. By using elfutils, many of its features such as relocation, reading of object files for many architectures, use of separate files with debugging information, etc, were leveraged, allowing the author to concentrate on the features that will be presented in the next sections.

Unless otherwise stated, the examples in the following sections use a Linux kernel image built for the x86-64 architecture from recent source code.[1]

The Linux kernel configuration option `CONFIG_DEBUG_INFO` has to be selected to instruct the compiler to insert the DWARF information. This will make the image much bigger, but poses no performance impact on the resulting binary, just like when building user space programs with debug information.

3.1 pahole

Poke-a-hole, the first dwarf, is used to find alignment holes in structs. It is the most advanced of all the tools in this project so far.

Architectures have alignment constraints, requiring data types to be aligned in memory in multiples of their word-size. While compilers do automatically align data structures, careful planning by the developer is essential to minimize the paddings (“holes”) required for correct alignment of the data structures members.

An example of a bad struct layout is in demand to better illustrate this situation:

```
struct cheese {
  char  name[17];
  short age;
  char  type;
  int   calories;
  short price;
  int   barcode[4];
};
```

Adding up the sizes of the members one could expect that the size of struct cheese to be 17 + 2 + 1 + 4 + 2 + 16 = 42 bytes. But due to alignment constraints the real size ends up being 48 bytes.

Using `pahole` to pretty-print the DWARF tags will show where the 6 extra bytes are:

```
/* <11b> ~/examples/swiss_cheese.c:3 */
struct cheese {
  char  name[17];    /*  0 17 */

  /* XXX 1 byte hole, try to pack */

  short age;         /* 18  2 */
  char  type;        /* 20  1 */

  /* XXX 3 bytes hole, try to pack */

  int   calories;    /* 24  4 */
  short price;       /* 28  2 */
```

[1] circa 2.6.21-rc5.

```
  /* XXX 2 bytes hole, try to pack */

  int   barcode[4]; /* 32 16 */
}; /* size: 48, cachelines: 1 */
   /* sum members: 42, holes: 3 */
   /* sum holes: 6 */
   /* last cacheline: 48 bytes */
```

This shows that in this architecture the alignment rule state that `short` has to be aligned at a multiple of 2 offset from the start of the struct, and `int` has to be aligned at a multiple of 4, the size of the word-size on the example architecture.

Another alignment rule aspect is that a perfectly arranged struct on a 32-bit architecture such as:

```
$ pahole long
/* <67> ~/examples/long.c:1 */
struct foo {
  int  a;     /*  0 4 */
  void *b;    /*  4 4 */
  char c[4]; /*  8 4 */
  long g;     /* 12 4 */
}; /* size: 16, cachelines: 1 */
   /* last cacheline: 16 bytes */
```

has holes when built on an architecture with a different word-size:

```
$ pahole long
/* <6f> ~/examples/long.c:1 */
struct foo {
  int   a;    /*  0 4 */

  /* XXX 4 bytes hole, try to pack */

  void *b;    /*  8 8 */
  char c[4]; /* 16 4 */

  /* XXX 4 bytes hole, try to pack */

  long g;     /* 24 8 */
}; /* size: 32, cachelines: 1 */
   /* sum members: 24, holes: 2 */
   /* sum holes: 8 */
   /* last cacheline: 32 bytes */
```

This is because on x86-64 the size of pointers and long integers is 8 bytes, with the alignment rules requiring these basic types to be aligned at multiples of 8 bytes from the start of the struct.

To help in these cases, `pahole` provides the `--reorganize` option, where it will reorganize the struct trying to achieve optimum placement regarding memory consumption, while following the alignment rules.

Running it on the x86-64 platform we get:

```
$ pahole --reorganize -C foo long
struct foo {
  int  a;     /*  0 4 */
  char c[4]; /*  4 4 */
  void *b;    /*  8 8 */
  long g;     /* 16 8 */
}; /* size: 24, cachelines: 1 */
   /* last cacheline: 24 bytes */
   /* saved 8 bytes! */
```

There is another option, `--show_reorg_steps` that sheds light on what was done:

```
$ pahole --show_reorg_steps
--reorganize -C foo long
/* Moving 'c' from after 'b' to after 'a' */
struct foo {
  int  a;     /*  0 4 */
  char c[4]; /*  4 4 */
  void *b;    /*  8 8 */
  long g;     /* 16 8 */
}; /* size: 24, cachelines: 1 */
   /* last cacheline: 24 bytes */
```

While in this case there was just one step done, using this option in more complex structs can involve many steps, that would have been shown to help understanding the changes performed. Other steps in the `--reorganize` algorithm includes:

- Combining separate bit fields
- Demoting bit fields to a smaller basic type when the type being used has more bits than required by the members in the bit field (e.g. `int a:1, b:2;` being demoted to `char a:1, b:2;`)
- Moving members from the end of the struct to fill holes
- Combining the padding at the end of a struct with a hole

Several modes to summarize information about all the structs in object files were also implemented. They will be presented in the following examples.

The top ten structs by size are:

```
$ pahole --sizes vmlinux | sort -k2 -nr
| head
hid_parser: 65784 0
hid_local: 65552 0
kernel_stat: 33728 0
module: 16960 8
proto: 16640 2
pglist_data: 14272 2
avc_cache: 10256 0
inflate_state: 9544 0
ext2_sb_info: 8448 2
tss_struct: 8320 0
```

The second number represents the number of alignment holes in the structs.

Yes, some are quite big and even the author got impressed with the size of the first few ones, which is one of the common ways of using this tool to find areas that could get some help in reducing data structure sizes. So the next step would be to pretty-print this specific struct, `hid_local`:

```
$ pahole -C hid_local vmlinux
/* <175c261>
        ~/net-2.6.22/include/linux/hid.h:300 */
struct hid_local {
  uint usage[8192];      // 0 32768
  // cacheline 512 boundary (32768 bytes)
  uint cindex[8192];     // 32768 32768
  // cacheline 1024 boundary (65536 bytes)
  uint usage_index;      // 65536 4
  uint usage_minimum;    // 65540 4
  uint delimiter_depth;  // 65544 4
  uint delimiter_branch;// 65548 4
}; /* size: 65552, cachelines: 1025 */
   /* last cacheline: 16 bytes */
```

So, this is indeed something to be investigated, not a bug in `pahole`.

As mentioned, the second column is the number of alignment holes. Sorting by this column provides another picture of the project being analyzed that could help finding areas for further work:

```
$ pahole --sizes vmlinux | sort -k3 -nr
| head
net_device: 1664 14
vc_data: 432 11
tty_struct: 1312 10
task_struct: 1856 10
request_queue: 1496 8
module: 16960 8
mddev_s: 672 8
usbhid_device: 6400 6
device: 680 6
zone: 2752 5
```

There are lots of opportunities to use `--reorganize` results, but in some cases this is not true because the holes are due to member alignment constraints specified by the programmers.

Alignment hints are needed, for example, when a set of fields in a structure are "read mostly," while others are regularly written to. So, to make it more likely that the "read mostly" cachelines are not invalidated by writes in SMP machines, attributes are used on the struct members instructing the compiler to align some members at cacheline boundaries.

Here is one example, in the Linux kernel, of an alignment hint on the struct `net_device`, that appeared on the above output:

```
/*
 * Cache line mostly used on receive
 * path (including eth_type_trans())
 */
  struct list_head poll_list
____cacheline_aligned_in_smp;
```

If we look at the excerpt in the `pahole` output for this struct where `poll_list` is located we will see one of the holes:

```
  /* cacheline 4 boundary (256 bytes) */
  void *dn_ptr;     /* 256  8 */
  void *ip6_ptr;    /* 264  8 */
  void *ec_ptr;     /* 272  8 */
  void *ax25_ptr;   /* 280  8 */

  /* XXX 32 bytes hole, try to pack */

  /* cacheline 5 boundary (320 bytes) */
  struct list_head poll_list;
     /* 320 16 */
```

These kinds of annotations are not represented in the DWARF information, so the current `--reorganize` algorithm can not be precise. One idea is to use the DWARF tags with the file and line location of each member to parse the source code looking for alignment annotation patterns, but this has not been tried.

Having stated the previous possible inaccuracies in the `--reorganize` algorithm, it is still interesting to use it in all the structs in an object file to present a list of structs where the algorithm was successful in finding a new layout that saves bytes.

Using the above kernel image the author found 165 structs where holes can be combined to save some bytes. The biggest savings found are:

```
$ pahole --packable vmlinux | sort -k4
-nr | head
vc_data          432   176 256
net_device      1664  1448 216
module         16960 16848 112
hh_cache         192    80 112
zone            2752  2672  80
softnet_data    1792  1728  64
rcu_ctrlblk      128    64  64
inet_hashinfo    384   320  64
entropy_store    128    64  64
task_struct     1856  1800  56
```

The columns are: struct name, current size, reorganized size and bytes saved. In the above list of structs only a few clearly, from source code inspection, do not have any explicit alignment constraint. Further analysis is required to verify if the explicit constraints are still needed after the evolution of the subsystems that use such structs, if the holes are really needed to isolate groups of members or could be reused.

The `--expand` option is useful in analyzing crash dumps, where the available clue was an offset from a complex struct, requiring tedious manual calculation to find out exactly what was the field involved. It works by "unfolding" structs, as will be shown in the following example.

In a program with the following structs:

```
struct spinlock {
  int magic;
  int counter;
};
```

```
struct sock {
  int protocol;
  struct spinlock lock;
};

struct inet_sock {
  struct sock sk;
  long        daddr;
};

struct tcp_sock {
  struct inet_sock inet;
  long cwnd;
  long ssthresh;
};
```

the `--expand` option, applied to the `tcp_sock` struct, produces:

```
struct tcp_sock {
  struct inet_sock {
    struct sock {
      int   protocol; /*  0 4 */
      struct spinlock {
        int magic;    /*  4 4 */
        int counter;  /*  8 4 */
      } lock;         /* 4 8 */
    } sk;             /* 0 12 */
    long    daddr;    /* 12 4 */
  } inet;             /* 0 16 */
  long      cwnd;     /* 16 4 */
  long      ssthresh; /* 20 4 */
}; /* size: 24 */
```

The offsets are relative to the start of the top level struct (`tcp_sock` in the above example).

3.2 pfunct

While `pahole` specializes on data structures, `pfunct` concentrates on aspects of functions, such as:

- number of goto labels
- function name length
- number of parameters
- size of functions
- number of variables

- size of inline expansions

It also has filters to show functions that meet several criteria, including:

- functions that have as parameters pointers to a struct
- external functions
- declared inline, un-inlined by compiler
- not declared inline, inlined by compiler

Also, a set of statistics is available, such as the number of times an inline function was expanded and the sum of these expansions, to help finding candidates for un-inlining, thus reducing the size of the binary.

The top ten functions by size:

```
$ pfunct --sizes vmlinux | sort -k2 -nr
| head
hidinput_connect: 9910
load_elf32_binary: 6793
load_elf_binary: 6489
tcp_ack: 6081
sys_init_module: 6075
do_con_write: 5972
zlib_inflate: 5852
vt_ioctl: 5587
copy_process: 5169
usbdev_ioctl: 4934
```

One of the attributes of the `DW_AT_subprogram` DWARF tag, that represents functions, is `DW_AT_inline`, which can have one of the following values:

- `DW_INL_not_inlined` – Neither declared inline nor inlined by the compiler
- `DW_INL_inlined` – Not declared inline but inlined by the compiler
- `DW_INL_declared_not_inlined` – Declared inline but not inlined by the compiler
- `DW_INL_declared_inlined` – Declared inline and inlined by the compiler

The `--cc_inlined` and `--cc_uninlined` options in `pfunct` use this information. Here are some examples of functions that were not explicitly marked as inline by the programmers but were inlined by gcc:

```
$ pfunct --cc_inlined vmlinux | tail
do_initcalls
do_basic_setup
smp_init
do_pre_smp_initcalls
check_bugs
setup_command_line
boot_cpu_init
obsolete_checksetup
copy_bootdata
clear_bss
```

For completeness, the number of inlined functions was 2526.

3.3 codiff

An object file diff tool, `codiff`, takes two versions of a binary, loads from both files the debugging information, compares them and shows the differences in structs and functions, producing output similar to the well known `diff` tool.

Consider a program that has a `print_tag` function, handling the following struct:

```
struct tag {
  int  type;
  int  decl_file;
  char *decl_line;
};
```

and in a newer version the struct was changed to this new layout, while the `print_tag` function remained unchanged:

```
struct tag {
  char type;
  int  decl_file;
  char *decl_line;
  int  refcnt;
};
```

The output produced by `codiff` would be:

```
$ codiff tag-v1 tag-v2
tag.c:
  struct tag |   +4
 1 struct changed
  print_tag  |   +4
 1 function changed, 4 bytes added
```

It is similar to the `diff` tool, showing how many bytes were added to the modified struct and the effect of this change in a routine that handles instances of this struct.

The `--verbose` option tells us the details:

```
$ codiff -V tag-v1 tag-v2
tag.c:
  struct tag |   +4
   nr_members: +1
    +int refcnt   /* 12 4 */
    type
     from: int    /*  0 4 */
     to:   char   /*  0 1 */
 1 struct changed
  print_tag  |   +4 # 29 → 33
 1 function changed, 4 bytes added
```

The extra information on modified structs includes:

- Number of members added and/or removed
- List of new and/or removed members
- Offset of members from start of struct
- Type and size of members
- Members that had their type changed

And for functions:

- Size difference
- Previous size -> New size
- Names of new and/or removed functions

3.4 ctracer

A class tracer, `ctracer` is an experiment in creating valid source code from the DWARF information.

For `ctracer` a method is any function that receives as one of its parameters a pointer to a specified struct. It looks for all such methods and generates kprobes entry and exit functions. At these probe points it collects information about the data structure internal state, saving the values in its members in that point in time, and records it in a relay buffer. The data is later collected in userspace and post-processed, generating html + CSS callgraphs.

One of the techniques used in `ctracer` involves creating subsets of data structures based on some criteria, such as member name or type. This tool so far just filters out any non-integer type members and applies the `--reorganize` code[2] on the resulting mini struct to possibly reduce the memory space needed for relaying this information to userspace.

One idea that probably will be pursued is to generate SystemTap [1] scripts instead of C language source files using kprobes, taking advantage of the infrastructure and safety guards in place in SystemTap.

3.5 pdwtags

A simple tool, `pdwtags` is used to pretty-print DWARF tags for data structures (struct, union), enumerations and functions, in a object file. It is useful as an example of how to use libdwarves.

Here is an example on the hello world program:

```
$ pdwtags hello
/* <68> /home/acme/examples/hello.c:2 */
int main(void)
{
}
```

This shows the “main” `DW_TAG_subprogram` tag, with its return type.

In the previous sections other examples of DWARF tag formatting were presented, and also tags for variables, function parameters, goto labels, would also appear if `pdwtags` was used on the same object file.

[2] Discussed in the `pahole` section.

3.6 pglobal

`pglobal` is an experimentation to print global variables and functions, written by a contributor, Davi Arnault.

This example:

```
$ cat global.c
int variable = 2;

int main(void)
{
  printf("variable=%d\n",
    variable);
}
```

would present this output with `pglobal`:

```
$ pglobal —ve hello
/* <89> /home/acme/examples/global.c:1 */
int variable;
```

which shows a list of global variables with their types, source code file, and line where they were defined.

3.7 prefcnt

`prefcnt` is an attempt to do reference counting on tags, trying to find some that are not referenced anywhere and could be removed from the source files.

4 Availability

The tools are maintained in a git repository that can be browsed at `http://git.kernel.org/?p=linux/kernel/git/acme/pahole.git`, and rpm packages for several architectures are available at `http://oops.ghostprotocols.net:81/acme/dwarves/rpm/`.

Acknowledgments

The author would like to thank Davi Arnault for pglobal, proving that libdwarves was not so horrible for a tool writer; all the people who contributed patches, suggestions, and encouragement on writing these tools; and Ademar Reis, Aristeu Rozanski, Claudio Matsuoka, Glauber Costa, Eduardo Habkost, Eugene Teo, Leonardo Chiquitto, Randy Dunlap, Thiago Santos, and William Cohen for reviewing and suggesting improvements for several drafts of this paper.

References

[1] Systemtap.
`http://sourceware.org/systemtap`.

[2] Ulrich Drepper. elfutils home page.
`http://people.redhat.com/drepper`.

[3] DWARF Debugging Information Format Workgroup. Dwarf debugging information format, December 2005. `http://dwarfstd.org`.