Some GCC Optimizations for Embedded Software

Khem Raj khem@himvis.com

Embedded Linux Conference 2014, San Jose, CA

Agenda

- Introduction
- What is GCC
- General Optimizations
- GCC specific Optimizations
- Embedded Processor specific Optimizations

GCC

- What is GCC – Gnu Compiler Collection
- Cross compiling
- Toolchain

Cross Compiler

- Cross compiling
  - Executes on build machine but generated code runs on different target machine
  - E.g. compiler runs on x86 but generates code for ARM
- Building Cross compilers
  - Crosstool-NG
  - OpenEmbedded/Yocto Project
  - Buildroot
  - OpenWRT
  - More ….

GCC Optimization Flags

- On
  - controls compilation time
  - Compiler memory usage
  - Execution speed and size/space
- O0
  - No optimizations
- O1 or O
  - General optimizations no speed/size trade-offs
- O2
  - More aggressive than O1
- Os
  - Optimization to reduce code size
- O3
  - May increase code size in favor of speed

GCC Optimization Levels

Property	General Opt level	Size	Debug info	Speed/ Fast
O	1	No	No	No
O1..O255	1..255	No	No	No
Os	2	Yes	No	No
Ofast	3	No	No	Yes
Og	1	No	Yes	No

Inline Assembly

- GCC inline assembly syntax

```
asm ( assembly template
      : output operands
      : input operands
      : A list of clobbered registers
    );
```

- Used when special instruction that gcc backends do not generate can do a better job
  - E.g. bsrl instruction on x86 to compute MSB
- C equivalent

```
long i;
for (i = (number >> 1), msb_pos = 0; i != 0; ++msb_pos)
  i >>= 1;
```

GCC Attributes/Built-ins

- Attributes aiding optimizations
  - Constant Detection
    - int __builtin_constant_p(exp)
  - Hints for Branch Prediction
    - __builtin_expect

```
#define likely(x)   __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

  - Prefetching
    - __builtin_prefetch
  - Align data
    - __attribute__ ((aligned (val)));
  - Packing Data
    - __attribute__((packed, aligned(val)))

GCC Attributes

- Pure functions
  - strcpy()
  - int __attribute__((pure)) static_pure_function([...])
- Constant functions
  - Special type of pure function with no side effects
  - strlen()
  - int __attribute__((const)) static_const_function([...])
- Restrict
  - void fn (int *__restrict__ rptr, int &__restrict__ rref)

Cache Optimization

```
#define L1_CACHE_CAPACITY (16384 / sizeof(int))
int array[L1_CACHE_CAPACITY][L1_CACHE_CAPACITY];
…
int main(void) {
 …

  for (i=0; i<L1_CACHE_CAPACITY; i++)
    for (j=0; j<L1_CACHE_CAPACITY; j++)
      array[j][i] = i*j;
...
}
```

```
#define L1_CACHE_CAPACITY (16384 / sizeof(int))
int array[L1_CACHE_CAPACITY][L1_CACHE_CAPACITY];

int main(void) {
  …
  for (i=0; i<L1_CACHE_CAPACITY; i++)
    for (j=0; j<L1_CACHE_CAPACITY; j++)
      array[i][j] = i*j;
 ...
}
```

Cache Optimizations

- 10x performance difference !!
  - Black Box Delta - 1:437454587
  - White Box Delta - 0:440943751
- Same number of Instructions but then why is difference ?
  - Memory access pattern changed
    - White example writes serially
    - Black example writes to cache line #0 and flushes it
  - Access pattern makes the whole difference

Data Cache Optimization

- Align Data to cache line boundary
  - int myarray[16] __attribute__((aligned(64)));
- Sequential data Access
  - Better use of loaded cache lines

Target Specific Optimizations

- CPU type
  - -march
- FPU utilization
  - X86/SSE, ARM/neon

Stack Optimizations

- Determine static stack usage
  - -fstack-usage
  - Information is in .su file

```
root@beaglebone:~# cat *.su
thrash.c:11:17:time_diff        16      static
thrash.c:25:5:main              24      static
```

- What contributes towards stack size
  - Local vars
  - Temporary data
  - Function parameters
  - Return addresses

Stack Optimizations – Help compiler

- Design it into Software
  - Avoid excessive Pre-emption
    - 2 concurrent tasks need more stack then two sequential processes
- Mindful use of local variable
  - Large stack allocation
    - Function scoped variables
    - E.g. operate on data in-place instead of making copies
    - Inline functions reduces stack usage
      - But not too-much
- Avoid long call-chains
  - Recursive functions

Stack Optimizations

- Use –Wstack-usage to get warned about stack usage

```
root@beaglebone:~# gcc thrash.c -Ofast -Wstack-usage=20
thrash.c: In function 'main':
thrash.c:42:1: warning: stack usage is 24 bytes [-Wstack-usage=]
```

- -fstack-check (specific to platforms e.g. Windows)
  - Adds a marker at an offset on stack
- -fconserve-stack
  - Minimize stack usage even if it means running slower

Size Optimizations

- Use Condensed Instructions Set
  - 16-bit instructions on 32-bit processors e.g. Thumb
  - -mthumb
- Abstract Functions
  - Compiler emit internal functions for common code
    - str* mem* built-in functions
- Multiple memory Access
  - Instructions which load/store multiple registers
    - LDM/STM (-Os in gcc)

Misc Optimizations

- -mslow-flash-data
  - Don't generate literal pool in code
  - GCC tries harder to synthesize constants
  - ARMv7-M/no-pic targets
- -mpic-data-is-text-relative
  - Assume data segment is relative to text segment on load
  - Avoids PC relative data relocation

Gold Linker

- Written from scratch in C++
- Targetted at ELF format
  - GNU ld was written for COFF and a.out (2-pass)
  - ELF format for retrofitted (needs 3 passes)
- Multi-threaded
- Supports ARM/x86/x86_64
  - Not all architectures supported by GNU ld are there yet
- Significant Speeds up link time for large applications
  - 5x in some big C++ applications

Gold Linker

- Configure toolchain to use gold
  - Add –enable-gold={default,yes,no} to binutils
- Coexists with GNU ld
  - Use gcc cmdline option
    - -fuse-ld=bfd – Use good'ol GNU ld
    - -fuse-ld=gold – Use Gold
  - While using LTO
    - -fuse-linker-plugin=gold
    - -fuse-linker-plugin=bfd
- Some packages do not _yet_ build with gold
  - U-boot, Linux kernel

Thanks

- Questions ?