# Some GCC Optimizations for Embedded Software

Khem Raj

Embedded Linux Conference 2014, San Jose, CA

# Agenda

- What is GCC
- General Optimizations
- GCC specific Optimizations
- Embedded Processor specific Optimizations
- Approaches to speed up compile time
- Additional tools

# 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

- O<n>
  - 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 |

# Aliasing

- Aliasing analysis is done for compiler to not optimize away aliased variables
- --fstrict-aliasing enabled at –O2 by default
- Use -Wstrict-aliasing for finding violations

# 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/Built-ins

- Pure functions
  - Value based on parameters and global memory only
  - strlen()
  - int __attribute__((pure)) static_pure_function([...])
- Constant functions
  - Special type of pure function with no side effects
  - Does not access global memory
  - strlen()
  - int __attribute__((const)) static_const_function([...])
- Restrict
  - void fn (int *__restrict__ rptr, int &__restrict__ rref)

# GCC Attributes/Built-ins

- Pragmas
  - Helpful when porting code written for other compilers
    - compilers ignore them if they are not understood
  - Avoid them if possible and use function/variable attributes instead
  - Eg. #pragma GCC optimize ("*string*"...)

# Cache Optimizations

```
#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/-mtune
    - Instruction scheduling
    - Considers CPU specific latencies
- FPU/SIMD utilization
  - X86/SSE, ARM/VFP/NEON etc.
- Target ABI specific
  - MIPS/-mplt
  - PPC/SPE
- Explore target specific options
  - gcc --target-help

# 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 )

# Profile Guided Optimizations

- -fprofile-generate
  - Phase I to generate data for feedback
- Run the instrumented code
  - Data is dumped to files
- -fprofile-use
  - Phase II Feedback data is used during optimization
- At the expense of doubling the compile-time

# Loop optimizations

- -funroll-loops
  - If compiler can determine N iterations
  - May generate faster code
  - Code-size will increase
- -funswitch-loops/-ftree-loop-im
  - Remove loop invariant code from loops
    - E.g. a constant assignment inside a loop
    - -funswitch-loops is for conditionals hoisting outside loop
- -fprefetch-loop-arrays
    - prefetching optimization
    - Know the L1/L2 cache sizes, line sizes

# Autovectorization/LTO

- -ftree-vectorize
- Some cases could regress the code
  - Indirect function calls in loop body
  - Switch operator inside loop
  - Help gcc with __builtin_assume_aligned
    - double *x = __builtin_assume_aligned(a, 16);
    - Qualify parameters with restrict keyword if they don't overlap
  - If expressions get complex vectorization may fail
- Link Time optimizations ( -flto )
  - Whole program optimized at link time

# Math related Optimizations

- -ffast-math
  - Speeds up math calculations at the expense of inaccuracy
- -fno-math-errno, -ffinite-math-only, -fno-signed-zeros
  - Also speed up math but no noise is introduced
- Sometimes better to use floats instead of doubles
  - E.g. on cortex-a8 single precision is faster

# 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

# Helpful binary utilities

- Disassemble
  - Compile source  with –g
  - Use objdump –d –S
    - Dump interleaved assembly and corresponding sources
- Dump ELF data
  - Readelf
  - Objdump
- Strings
  - Display printable strings in file
- Nm
  - List sybols from objects/binaries
- Size
  - Display size of sections in binary/objects
- Addr2line
  - Convert addresses into linenumber:filename

# Takeaways

- Help the compiler and it will help you
- Know the target hardware
- Resource Limitations (CPU, Memory, slow I/O, Power)
- Measure first optimize later
- Use tools like oprofile,gcov, gprof, valgrind, perftools
- Perfect is enemy of good

# Thanks

- Questions ?