Stories from the compiler engineering front

Christoph Müllner

The solution is embedded.
Who are we

Embedded design house

Complete solutions for embedded systems (HW, BSP and application)
Building toolchains, cross-compilation, static code analysis, DSLs,
parser generators, custom language runtime, language bindings etc.

Compiler/tools engineering (compiler front)

GCC, LLVM, OpenJDK, [glibc, Linux kernel, jemalloc, openssl] …
Previously: CACAO (Java JIT), HHVM (Facebook’s PHP-JIT, now Hack only)

Current main target architecture: AArch64 (64-bit ARM)
What we do (at the compiler front)

**Measure**
SPEC CPU, SPECjbb, dhrystone, coremark, synthetic benchmarks, phpbench…
Compare CPU cores/architectures, run single-threaded or scaling-up, ...

**Analyse**
Benchmark results, perf reports, reading disassembly, creating microbenchmarks, reading processor manuals, ask the CPU designer

**Improve**
Eliminate performance bottlenecks, improve cache utilization, fix compiler bugs
The aarch64 ISA specification allows a left shift amount to be applied after extension in the range of 0 to 4 (encoded in the imm3 field).

This is true for at least the following instructions:

* ADD (extend register)
* ADDS (extended register)
* SUB (extended register)

The result of this patch can be seen, when compiling the following code:

```c
uint64_t myadd(uint64_t a, uint64_t b)
{
    return a+(((uint8_t)b)<<4);
}
```

Without the patch the following sequence will be generated:

```
0000000000000000 <myadd>:
 0: d37c1c21 ubfiz x1, x1, #4, #8
 4: 8b000020 add x0, x1, x0
 8: d65f03c0 ret
```

With the patch the ubfiz will be merged into the add instruction:

```
0000000000000000 <myadd>:
 0: 8b211000 add x0, x0, w1, uxtb #4
 4: d65f03c0 ret
```

Tested with "make check" and no regressions found.
Working with GCC

- CPU maintenance in **AArch64 architecture backend**
  - Adjusting instruction **cost table** (challenges: GCC’s cost model vs. reality)
  - **Instruction scheduling** (challenges: out-of-order execution, speculation, multiple issue)
  - CPU specific optimization defaults (function alignment)

- Maintenance of **ILP32** for AArch64 (arm64)
  - `sizeof(long) == 4` (ILP32) vs. `sizeof(long) == 8` (LP64)
  - Irrelevant for most applications
  - Much smaller memory usage
  - Better cache utilization
  - **7-10 % performance gain** on average
  - Changes in GCC, glibc, Linux kernel, jemalloc
Improve cache utilization

- High-end CPUs have **data caches** ("memory hierarchy")
- Load data: data must be in cache, if it’s not there it must be transferred there
- Transfer in blocks ("cache lines")
- Let’s assume a cache line size of **64 bytes**
- **Cache hit**: load costs 1 cycle (0.3 ns @ 3.0 GHz)
- **Cache miss**: load costs ~300 cycles (100 ns @ 3.0 GHz)
- Cache is limited -> **cache line eviction**
- What can be done to **improve cache utilization**?
  - Use as many bits of a cache line as possible
Improve cache utilization: structs/records

- Data often stored as array/list of structs (or records)
- **Hot loop**: iteration over array/list of structs
- Access to different **fields** in each loop iteration
- **Worst case**: we need to get a cache line for a single byte/bit read
- **Best case**: we need all bytes of a cache line
- Programmers should optimise...
- But compilers could do as well...

```c
struct message {
    [... 63 bytes ...]
    uint8_t is_urgent;
};
struct message msgs[ARRAY_SIZE];

for (i=0; i<ARRAY_SIZE; i++) {
    is_urgent |= msgs[i].is_urgent;
}
```

Struct reorg transformations: field reordering

- Works for large structs (bigger than one cache line)
- Optimisation 1: order fields by hotness
- Optimisation 2: order fields by access order (PGO)
- No additional costs
- No changes of allocation and access sites required

```c
struct msg {
    int index;
    [... 64 bytes ...]
    uint64_t ctime;
};

int cmp_le_msg(a, b) {
    if (a->ctime < b->ctime)
        return true;
    if (a->ctime > b->ctime)
        return false;
    return (a->index <= b->index);
}
```

```c
struct msg {
    int index;
    uint64_t create_time;
    [... 64 bytes ...]
};
```
Struct reorg transformations: packing

- Packing structs to improve data density
- Works if unaligned access is no problem on the target machine
- Improves cache line utilisation for sequential processing (e.g. array iteration)
- GCC offers `-fpack-struct=N` (N...max. alignment)
- Caution: `-fpack-struct` is dangerous (no escape analysis)

```c
struct s {
    uint8_t a; //offset 0
    void* b;   //offset 8
    void8_t c; //offset 16
}; //size 24
```

```c
struct s {
    uint8_t a; //offset 0
    void* b;   //offset 1
    void8_t c; //offset 9
}; //size 10
```
Struct reorg transformations: padding

- Aligning structs to decrease cache line crossing
- Alignment to cache line
- Each struct starts at beginning of cache line
- In array of structs: average size of struct increases
- Improves cache line utilisation for non-sequential processing (e.g. linked lists)

```c
struct s {
    uint8_t a[40];  //offset 0
    struct s* next; //offset 8
}; //size 48
```

```c
struct s {
    uint8_t a[40];  //offset 0
    struct s* next; //offset 8
}; //size 64
```
Struct reorg transformations: drop fields

- Abandon unused fields (hey programmer, are you listening?)
- Less memory footprint
- Less cache pollution

```c
struct node {
    char A[];
    char B[]; //unused
    long C, D, E, F, G, H;
    double M;
};
```

```c
struct node {
    char A[];
    long C, D, E, F, G, H;
    double M;
};
```
Struct reorg in compilers

- Compilers optimise (=change) code for improved efficiency
- We must maintain correctness
  - Data layout of struct needs to be equal for all access locations
- We must follow interoperability rules (calling convention, etc.)
  - Not so strict inside a compilation unit (e.g. static inline functions might not exist)
  - LTO: whole program is compilation unit
  - “That’s cheating!” - Yes, but every compiler does it...
- What information is needed to let a compiler do struct reorg?
  - Which struct can be reorganized? => escape analysis
  - How should a struct be reorganized? => profile guided optimization
- Escape analysis: does one instance of a given type leave a compilation unit
- Recognizing all field accesses is not trivial (pointers, casting, aliasing etc.)
#define MYCONST 3.14f

int mycmp(float x) {
    if ((MYCONST / x) < 0)
        return 1;
    return 0;
}

- MYCONST > 0
- no infinites ("fastmath")

#define MYCONST 3.14f
int mycmp(float x) {
    if (x < 0)
        return 1;
    return 0;
}

mycmp(float):
    fcmpe   s0, #0.0
    cset w0, mi
    ret

mycmp(float):
    mov w0, 62915
    movkw0, 0x4048, lsl 16
    fmov s1, w0
    fdiv s0, s1, s0
    fcmpe s0, #0.0
    cset w0, mi
    ret
Contact

Feel free to contact me:

christoph.muellner@theobroma-systems.com

If you want to join us:

careers@theobroma-systems.com

Our website:

www.theobroma-systems.com