Performance Tuning Dive-In

Profiling, Compiling and Libraries

Marat Dukhan

September 24, 2013

Plan

3 compilers you should know

- GNU Compilers Collection (gcc/g++)
  - Default compiler on Linux
  - Produces very fast code
- Clang C/C++ Compilers (clang/clang++)
  - Default compiler on Mac OS X and FreeBSD
  - Much better error reporting than gcc
  - Usually produces slower code than gcc, but sometimes can be faster
  - Compiler options, pragmas, and other extensions compatible with gcc
- Intel C/C++ Compilers (icc/icpc)
  - Better error reporting than gcc
  - For Intel processors often produces faster code than gcc
  - More or less compatible with gcc

Basic optimization options

- `-Og` enable only optimizations which do not mess up debugging
- `-O2` enable all basic optimizations for performance
- `-O3` = `-O2` + speculative optimizations
- `-Of` = `-O3` + unsafe floating-point optimizations
  - $a + b + c + d = (a + b) + (c + d)$
- `-Os` optimize for code size
  - For large code bases this results in faster code than `-O3`

Machine-specific optimization options (gcc and clang)

- `-march=<cpu>` use all instructions available on `<cpu>`. Valid `<cpu>` values:
  - `core2` for Intel Core 2 and Xeon processors of the same generation (Harpertown, Clovertown). For 45nm Core 2. processors (and Harpertown Xeons) you should also specify `-msse4.1` because they additionally support SSE4.1 instruction set.
  - `corei7` for Intel Nehalem and Westmere: first generation Core-i7 processors and Xeon 5500 and 5600 series (like Jinx!).
  - `corei7-avx` for Intel Sandy Bridge processors (aka second-generation Core-i7).
  - `core-avx-i` for Intel Ivy Bridge processors (aka third-generation Core-i7).
  - `core-avx2` for Intel Haswell processors (aka fourth-generation Core-i7).
  - `bdver1` for AMD Bulldozer processors (Opteron 3200, 4200, and 6200 series).
  - `bdver2` for AMD Piledriver processors (Trinity APUs, Opteron 3300, 4300, and 6300 series).

Machine-specific optimization options (gcc and clang)

Continued

- `-march=<cpu>` use all instructions available on `<cpu>`. Valid `<cpu>` values:
  - `amdfam10` for AMD K10 (most pre-buildozer CPUs from AMD, including Opteron 4100 and 6100 series and Phenom processors).
  - `atom` for Intel Atom processors.
  - `btver1` for AMD Bobcat processors (E-350 and alike).
  - `btver2` for AMD Jaguar processors (A4-5000 and alike).
  - `native` for the CPU you compile on (sometimes misdetects CPU).
- `-mtune=<cpu>` option instructs the compiler to tune the code for specific processor
  - Accepted `<cpu>` values are the same as for `-march` option
  - If this option is not specified, program is tuned for CPU specified in `-march=<cpu>`

Inter-Procedural Optimization

- Postpones code generation until linking stage when all information about the program is available.
- Different options for different compilers:
  - `-ipo` for Intel Compiler.
  - `-flto` for GNU and Clang compilers.
  - These options must be specified during both compilation and linking.

PGO Overview

Often there is not enough information in compilation time to properly optimize the code.
Profile-Guided Optimization allows the compiler to collect information from program runs, and then use this information to improve optimization quality.

Using PGO with GNU Compilers

1. Compile your program with `-fprofile-generate` parameter

   ```
   gcc ${CFLAGS} -fprofile-generate -o myapp myapp.c
   ```

2. Run the program on representative input

   ```
   ./myapp testdata.input
   ```

3. Compile your program again with `-fprofile-use` parameter

   ```
   gcc ${CFLAGS} -fprofile-use -o myapp myapp.c
   ```

Using PGO with Intel Compilers

1. Compile your program with `-prof-gen` parameter

   ```
   gcc ${CFLAGS} -prof-gen -o myapp myapp.c
   ```

2. Run the program on representative input

   ```
   ./myapp testdata.input
   ```

3. Compile your program again with `-prof-use` parameter

   ```
   gcc ${CFLAGS} -prof-use -o myapp myapp.c
   ```

Plan

Intel MKL

Intel MKL (Math Kernel Library) provides many primitives for scientific computing:

1. Linear Algebra (dense and sparce)
2. Nonlinear Optimization
3. PDE Solvers
4. Fast Fourier Transform (FFT)
5. Random Number Generation
6. Vector Elementary Functions

Installed on Jinx in `/opt/intel/mkl/`

Intel IPP

Intel IPP (Integrated Performance Primitives) library is focused on media processing:

1. Image processing (e.g. resizing and convolution)
2. Computer vision (e.g. Canny Edge)
3. Image coding primitives (e.g. JPEG, PNG)
4. Video coding primitives (e.g. MPEG-2 and H.264)
5. Audio coding (e.g. MPEG-2 and H.264)
6. Compression (e.g. deflate and BWT)
7. Small matrix operations (for 3D Graphics and Ray Tracing)

Installed on Jinx in `/opt/intel/Compiler/11.1/059/ipp/`

ATLAS

ATLAS = Automatically Tuned Linear Algebra Software.

1. Implements full BLAS interface
2. Takes hours to days to set up
3. Usually slower than Intel MKL
4. Portable to new architectures

Website: math-atlas.sourceforge.net

FFTW

FFTW = Fastest Fourier Transform in the West.

1. FFTW is a multi-threaded and SIMD-optimized library for one-dimensional and multi-dimensional Fourier transform.
2. The recent version is capable of doing distributed FFT via MPI.
3. FFTW is slower than Intel MKL, but is more portable.
4. Intel MKL supports FFTW interface.

Website: fftw.org
Paper: **M. Frigo and S. G. Johnson (2005).** *The Design and Implementation of FFTW3, Proceedings of the IEEE.*

Yeppp!

Yeppp! is a fast cross-platform library for vector operations

Yeppp! is developed by Marat Dukhan and provides high-performance vector operations for many platforms (Windows, Android, Mac OS X, GNU/Linux), architectures (x86, x86-64, Xeon Phi, ARM, MIPS) and programming languages (C, C++, C#, FORTRAN, Java).
Website: www.yeppp.info

Elemental

Elemental is a distributed-memory dense linear algebra library.

Elemental is developed by Jack Poulson, who will join the School of CSE in November, and allows efficient manipulations on matrices and vectors distributed across MPI nodes.
Website: www.libelemental.org
Paper: **J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero (2013).** *Elemental: A new framework for distributed memory dense matrix computations, ACM Transactions on Mathematical Software.*

Plan

Introduction

Modern processors and operating systems have performance counters, which track performance-affecting events, such as the number of context switches or cache misses.
Linux provides `perf_event` subsystem which allows to read these counters and use them to hunt performance problems. `perf` utility makes it even easier.
In fact, `perf` is a collection of utilities:

- `perf list` lists available performance counters
- `perf top` reports system-wide statistics on performance events
- `perf stat` runs a program and reports the total number of performance events for this program.
- `perf record` runs a program and records a detailed information about performance events.
- `perf report` allows to browse information collected by perf record.

Performance Events

Here are some of the performance events from Jinx node (`perf list`):

- `cycles` counts the number of CPU cycles.
- `instructions` counts the number of instructions.
- `cache-references` counts the number of executed load, store, and prefetch instructions.
- `cache-misses` counts the number of load, store, and prefetch instructions which could not get data from cache.
- `branch-instructions` counts the number of executed branch instructions (think if, else, while, for).
- `branch-misses` counts the number of branch instructions which were mispredicted by processor.
- `stalled-cycles-frontend` counts the number of wasted cycles due problem in instruction decoder.
- `stalled-cycles-frontend` counts the number of wasted cycles due problem in execution core.

Summary Statistics

To get summary statistics for a program run use commend

```
perf stat -e <counter1> myapp args...
```

If you specify `instructions` and `cycles`, perf will compute average number of instructions per cycle (IPC):

```
perf stat -e instructions -e cycles myapp args...
```

Targets for IPC values:

- The higher is IPC the better.
- Maximum possible IPC on Jinx nodes is 4 (5 in rare cases).
- Most codes can achieve at least 2 IPC. Think of it as your target.
- If IPC is less than 1, you need to work on it.

Summary Statistics
Continued

If you specify `branches` and `branch-misses`, perf will compute percent of mispredicted branches:

```
perf stat -e branches -e branch-misses myapp args...
```

Targets for percent of mispredicted branches:

- The less branches are mispredicted the better
- If more than 5% of branches are mispredicted, you need to look at it.

If you specify `cache-references` and `cache-misses`, perf will compute percent of cache misses:

```
perf stat -e cache-references -e cache-misses myapp args...
```

Targets for percent of cache misses:

- The less cache references are missed the better
- If more than 10% of cache references are missed, investigate it.

Detailed Statistics

To collect detailed statistics run `perf record` utility. If you investigate problems with some specific metric, you may specify performance counter with `-e` parameter:

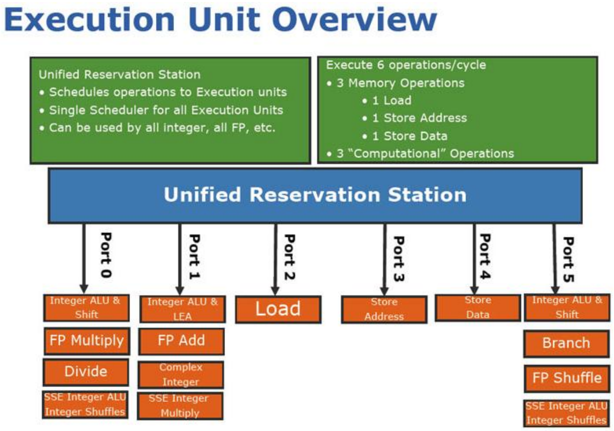
```
perf record -e branch-misses myapp args...
```

`perf record` will create `perf.data` file with performance measurements. You may browse this file with `perf report` utility:

```
perf report
```

Introduction

Intel Architecture Code Analyser is a static analysis tool which suggests how instructions in your code are scheduled to execution units inside the CPU.


Execution Unit Overview
Unified Reservation Station
• Schedules operations to Execution units
• Single Scheduler for all Execution Units
• Can be used by all integer, all FP, etc.
Execute 6 operations/cycle
• 3 Memory Operations
• 1 Load
• 1 Store Address
• 1 Store Data
• 3 "Computational" Operations
Unified Reservation Station
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
Integer ALU & Shift
FP Multiply
Divide
SSE Integer ALU Integer Shuffles
Integer ALU & LEA
FP Add
Complex Integer
SSE Integer Multiply
Load
Store Address
Store Data
Integer ALU & Shift
Branch
FP Shuffle
SSE Integer ALU Integer Shuffles

How to Use IACA

1. Include IACA header into your C or C++ code

   ```
   #include <iacaMarks.h>
   ```

2. Enclose the region of interest in `IACA_START` and `IACA_END` macros
   - Limitation: region of interest must be linear block of code (i.e. without if/else statements, loops, function calls)

3. Compile your source into object file

   ```
   gcc -c ${CFLAGS} -o algo.o algo.c
   ```

4. Run IACA

   ```
   iaca -64 -arch WSM -o algo.log algo.o
   ```

Example of Using IACA

```
#include <algorithm>
#include <iacaMarks.h>

using namespace std;

void filter(int *a, int *b, int *c, int *m, unsigned n) {
    for (unsigned i = 0; i < n; i++) {
        IACA_START
        m[i] = a[i] * b[i] * 8732 +
            c[i] * 184 +
            a[i] * 99 +
            b[i] * 321;
        IACA_END
    }
}
```

Example IACA Output

```
Throughput Analysis Report
--------------------------
Block Throughput: 5.05 Cycles       Throughput Bottleneck: Port1

Port Binding In Cycles Per Iteration:
-------------------------------------------------------------------------
|  Port  |  0   -  DV  |  1   |  2   -  D   |  3   -  D   |  4   |  5   |
-------------------------------------------------------------------------
| Cycles | 2.5    0.0  | 5.0  | 3.0    3.0  | 1.0    0.0  | 1.0  | 2.5  |
-------------------------------------------------------------------------

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

| Num Of |              Ports pressure in cycles               |    |
|  Uops  |  0  - DV  |  1  |  2  -  D  |  3  -  D  |  4  |  5  |    |
---------------------------------------------------------------------
|   1    |           |     | 1.0   1.0 |           |     |     |    | mov r10d, dword ptr [rdi+rax*4]
|   1    |           |     | 1.0   1.0 |           |     |     |    | mov r9d, dword ptr [rsi+rax*4]
|   1    |           |     | 1.0   1.0 |           |     |     |    | mov ebx, dword ptr [rdx+rax*4]
|   1    | 0.5       |     |           |           |     | 0.5 |    | mov r11d, r10d
|   1    |           | 1.0 |           |           |     |     | CP | imul r11d, r9d
|   1    |           | 1.0 |           |           |     |     | CP | imul ebx, ebx, 0xb8
|   1    |           | 1.0 |           |           |     |     | CP | imul r9d, r9d, 0x141
|   1    |           | 1.0 |           |           |     |     | CP | imul r11d, r11d, 0x221c
|   1    | 0.5       |     |           |           |     | 0.5 |    | add r11d, ebx
|   1    | 0.4       |     |           |           |     | 0.6 |    | mov ebx, 0x63
|   1    |           | 1.0 |           |           |     |     | CP | imul r10d, ebx
|   1    | 0.4       |     |           |           |     | 0.6 |    | add r10d, r11d
|   1    | 0.6       |     |           |           |     | 0.4 |    | add r9d, r10d
|   2    |           |     |           | 1.0       | 1.0 |     |    | mov dword ptr [rcx+rax*4], r9d
Total Num Of Uops: 15
```

Introduction

Valgrind is a framework which allows to intercept execution of specific classes of instructions (like loads and stores). There are a number of tools built on top of valgrind for research and analysis. Cachegrind utility simulates processor caches and provides information on the number of cache misses.
Unlike `perf` utility, cachegrind allows to change size and other parameters of simulated caches and analyse how it affects cache miss rate.

Using Cachegrind

A program myapp can be launched under cachegrind with command

```
valgrind --tool=cachegrind myapp args...
```

Cachegrind parameters `-I1`, `-D1`, and `-LL` allow to change parameters of simulated caches:

```
valgrind --tool=cachegrind --D1=65536,8,64 myapp args...
```

Plan

Summary

- Smart use of compilers and performance libraries allow to improve performance without putting much effort into it.
- Profilers will hint you where to start optimizing when more extensive performance tuning is needed.