About me

- Andrey Bokhanko
- Architect and project lead of the Intel Compiler for Itanium
- Started to work on optimizing compilers back in 1999
- Before that, worked in Elbrus, in the compiler team
- PhD thesis on… you guessed it – compilers (register allocation)
Scope of this lecture

- Introduction to *optimizing compilation*
- “Optimizing compilation” – what is it? What is it about?
  - Mostly not on compilers themselves
- Not an introduction to compiler construction – 1 hour is definitely **not** enough for this
- A view from industrial perspective
  - Sorry, I’m not very knowledgeable in state of things in academia
What is a compiler?

- “…is a computer program that transforms source code … into another computer language (object code)” (wikipedia)
- An *essential* tool – you can’t write computer programs without it
- An “invisible” tool
- Speed of compilation and stability is essential
What is an optimizing compiler?

- Does everything that a non-optimizing compiler should do
- Also, tries to optimize a compiled program, while leaving its semantic intact
  - Not as easy as it might seem
- Frankly, is an optional, not essential tool
- Might bring valuable competitive advantage to
  - Software developer
  - OS vendor
  - Hardware vendor
What “optimize” means?

- Make user program faster, smaller, consume less power
  - “Traditional” optimizing compilers concentrate on “faster” part
  - “Smaller” is important only for embedded systems
  - Power-efficiency is extremely important in mobile systems. However, tools are in their infancy
  - Compilation for GPU uses distinct approaches; there is a trend for them to become more general
- Preserving semantic of a user program is a must
Who creates optimizing compilers?

- This is a complex and expensive task
- Usually, two types of vendors pay for this work
  - Hardware vendors
    - They want user programs to run fast on their hardware
  - OS vendors
    - They want user programs to run fast on their OSes
Benchmarking

- How to measure if a compiler speed-ups user programs?
  - Easy for a single program, single input, single system
  - Not so easy for multiple programs, compilers, OSes, machines
- We want to simulate and measure what users typically do on a system
Benchmarking, cont

- Should be done on a set of programs / workloads
  - Representing what customers are most likely going to run on a system
- Should be reproducible
- Overall performance is a combination of
  - Machine
  - OS (including libraries)
  - Compiler
Standard benchmarks

- **SPEC**: Standard Performance Evaluation Corporation
  - Many benchmarks; most important is SPEC CPU
  - [www.spec.org](http://www.spec.org)

- **TPC**: Transaction Processing Performance Council
  - Performance of transaction processing systems (databases)
  - [www.tpc.org](http://www.tpc.org)

- **EEMBC**: Embedded Microprocessor Benchmark Consortium
  - Benchmarks for embedded/mobile systems
  - Allow some degree of source code modification
  - [www.eembc.org](http://www.eembc.org)

- Kernels, toy programs, synthetic benchmarks (Dhrystone, Whetstone) and MIPS numbers were popular, not anymore
Standard benchmarks, cont
SPEC CPU

- Probably the most important benchmark for general-purpose computing
  - Though biased towards technical and scientific computing
- First version is SPEC CPU92
- Current version is SPEC CPU2006
- SPECv6 is under development
- Aims to be vendor- and platform-neutral
  - All major players are members, try to influence SPEC development
- Seriously impacts ASP, especially for server machines
- Makes or breaks careers
What’s inside SPEC CPU2006

- CINT2006
  - 400.perlbench (C, programming language)
  - 401.bzip2 (C, compression)
  - 403.gcc (C, C compiler)
  - 429.mcf (C, combinatorial optimizations)
  - 445.gobmk (C, artificial intelligence: go)
  - 456.hmmer (C, search gene sequence)
  - 458.sjeng (C, artificial intelligence: chess)
  - 462.libquantum (C, physics / quantum computing)
  - 464.h264ref (C, video compression)
  - 471.omnetpp (C++, discrete event simulation)
  - 473.astar (C++, path-finding algorithms)
  - 483.xalancbmk (C++, XML processing)
What’s inside SPEC CPU2006, cont

- CFP2006
  - 410.bwaves (Fortran, fluid dynamics)
  - 416.gamess (Fortran, quantum chemistry)
  - 433.milc (C, physics / quantum chromodynamics)
  - 434.zeuscmp (Fortran, physics / CFD)
  - 435.gromacs (C / Fortran, biochemistry)
  - 436.cactusADM (C / Fortran, physics)
  - 437.leslie3d (Fortran, fluid dynamics)
  - 444.namd (C++, biology)
  - 447.deall (C++, finite element analysis)
  - 450.soplex (C++, linear programming, optimization)
  - 453.povray (C++, image ray-tracing)
  - 454.calculix (C / Fortran, structural mechanics)
  - 459.GemsFDTD (Fortran, computational electromagnetics)
  - 465.tonto (Fortran, quantum chemistry)
  - 470.lbm (C, fluid dynamics)
  - 481.wrf (C / Fortran, weather)
  - 482.sphinx3 (C, speech recognition)
How to measure results?

- Should we just summarize execution time of all tasks?
- In reality, execution time got normalized to a reference time (obtained on some old machine)
There are two kinds of scores:

- speed: single copy of each task
- rate: multiple copies of each task (usually equal to the number of cores)

Also, two kinds of measurements:

- base: same options for all tasks
- peak: different options allowed for different asks

Total score = geomean of all individual scores

Reported separately for CINT and CFP
SPEC CPU, cont

Hewlett-Packard Company
HP Integrity rx6600 (1.6GHz/24MB Dual-Core Intel Itanium 2)

CPU2006 license: 03
Test sponsor: Hewlett-Packard Company
Tested by: Hewlett-Packard Company

SPECint®2006 = 15.7
SPECint base2006 = 14.5

Test date: Aug-2006
Hardware Availability: Sep-2006
Software Availability: Sep-2006

SPECint2006 = 15.7
### Hardware

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Name</td>
<td>Dual-Core Intel Itanium 2 9050</td>
</tr>
<tr>
<td>CPU Characteristics</td>
<td>1.6GHz/24MB, 533MHz FSB</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>1600</td>
</tr>
<tr>
<td>FPU</td>
<td>Integrated</td>
</tr>
<tr>
<td>CPU(s) enabled</td>
<td>2 cores, 1 chip, 2 cores/chip</td>
</tr>
<tr>
<td>CPU(s) orderable</td>
<td>1-4 chips</td>
</tr>
<tr>
<td>Primary Cache</td>
<td>16 KB I + 16 KB D on chip per core</td>
</tr>
<tr>
<td>Secondary Cache</td>
<td>1 MB I + 256 KB D on chip per core</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>12 MB I+D on chip per core</td>
</tr>
<tr>
<td>Other Cache</td>
<td>None</td>
</tr>
<tr>
<td>Memory</td>
<td>24 GB (24x1GB DIMMs)</td>
</tr>
<tr>
<td>Disk Subsystem</td>
<td>73GB 10K RPM SAS</td>
</tr>
<tr>
<td>Other Hardware</td>
<td>None</td>
</tr>
</tbody>
</table>

### Software

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>HPUX11i-TCOE B.11.23.0609</td>
</tr>
<tr>
<td>Compiler</td>
<td>HP C/C++ Developer's Bundle</td>
</tr>
<tr>
<td>Auto Parallel</td>
<td>No</td>
</tr>
<tr>
<td>File System</td>
<td>vxfs</td>
</tr>
<tr>
<td>System State</td>
<td>Multi-user</td>
</tr>
<tr>
<td>Base Pointers</td>
<td>32-bit</td>
</tr>
<tr>
<td>Peak Pointers</td>
<td>32-bit</td>
</tr>
<tr>
<td>Other Software</td>
<td>MicroQuill Smartheap 8.0</td>
</tr>
</tbody>
</table>
How to optimize?

- Basically, two ways:
  - Eliminate redundant / slow computations
    - Classic optimizations
  - Keep execution resources busy
    - Especially important for statically-scheduled machines
  - Sometimes, these two goals conflict with each other
Elimination of redundant / slow computations

- Most classic optimizations
  - Dead code elimination
  - Common subexpression elimination
  - Constant folding
  - Strength reduction
  - ...

- Generally, help everywhere, so implemented everywhere

- Known for very, very long time
Dead code elimination

int foo(int x, int y) {
    int z = x / y;
    return x * y;
}

int foo(int x, int y) {
    return x * y;
}
Common subexpression elimination

```
int foo( int x, int y) {
    int z1 = x * y + 1;
    int z2 = x * y + 2;
    return z1 + z2;
}
```

```
int foo( int x, int y) {
    int t = x * y;
    int z1 = t + 1;
    int z2 = t + 2;
    return z1 + z2;
}
```
CSE + constant folding + DCE

int foo( int x, int y) {
    int z1 = x * y + 1;
    int z2 = x * y + 2;
    return z1 + z2;
}

int foo( int x, int y) {
    int t = x * y;
    return t + t + 3;
}
int foo(char *A) {
    for (int i = 0; i < 100; i++) {
        *((int *)((A + i) * 4)) = 0;
    }
}

int foo(char *A) {
    char *t;
    for (int i = 0, t = A + i;
         i < 100; i++) {
        *((int *)t) = 0;
        t += 4;
    }
}
int foo( int x) {
    return x * 2;
}

int foo( int x) {
    return x << 1;
}
Keeping execution resources busy

- When execution resources may lay unused?
  - Parallel machine with only some of available execution resources used
  - Waiting for a dependency
    - Especially memory dependency!
- Advanced, aggressive, speculative scheduling
- Memory optimizations
- Often implemented in hardware, especially in OOO machines
Advanced, aggressive, speculative scheduling

- Scheduling is reordering of instructions in order to keep execution units busy all the time.
- Advanced = using advanced techniques, like copying of instructions.
- Aggressive = global in scope.
- Speculative = executing instructions that might not be executed.
Control speculation

```c
int foo( int *x) {
    if (x != 0) {
        return *x;
    }
    return 0;
}
```

```c
int foo( int *x) {
    int t = *x;
    if (x != 0) {
        return t;
    }
    return 0;
}
```
Data speculation

```
int foo( int *x, int *y) {
    if (x != 0) {
        *y = 0;
        return *x;
    }
    return 0;
}
```

```
int foo( int *x, int *y) {
    int t = *x;
    if (x != 0) {
        *y = 0;
        return t;
    }
    return 0;
}
```
Unrolling

```c
int foo( char *A) {
    char *t = A + i;
    for (int i = 0; i < 100; i++) {
        *((int *)t) = 0;
        t += 4;
    }
}
```

```c
int foo( char *A) {
    char *t1, *t2;
    for (int i = 0,
         t1 = A + i, t2 = A + i + 4;
         i < 50; i++) {
        *((int *)t1) = 0;
        t1 += 8;
        *((int *)t2) = 0;
        t2 += 8;
    }
}
```
Prefetching

```c
int foo( int *A) {
    int x = 0;
    for (int i = 0; i < 100; i++) {
        x += A[i];
    }
    return x;
}

int foo( int *A) {
    int x = 0;
    for (int i = 0; i < 100; i++) {
        x += A[i];
        prefetch A[i + 4];
    }
    return x;
}
```
Automatic vectorization and parallelization

- Usually, a developer uses vector instructions with assembler / intrinsics / libraries
  - Optimizing compilers can employ automatic “vectorization”

- Same with coarse-grained (core-grained) parallelism
  - Usually employed with libraries
  - Compiler able to automatically parallelize
    - Very hard to do; sometimes hurts performance
Profiling

- It is important to know *where* and *how* to optimize
- Instrumentation of user program, then “profile collection” run
- Several deficiencies
  - Transforms compilation into two-step process
  - How to choose input for profile collection run? What if it is not representative?
Future developments

- Power usage Power optimizations
  - How to collect power usage data? Power profiling?
- Dynamic optimizations
  - Binary compilers
  - Virtual machines
  - Run-time optimizations (LLVM)
- Optimization for graphic, heterogeneous computing
  - Specialized APIs / languages used now
- We count on you! I’m too old for this… 😊
What to read

- PLDI, CGO conferences
Thank you for your time!

- Questions?
- andreybokhanko@gmail.com