Investigation of the improved performance on Haswell processors

Marco Guerri

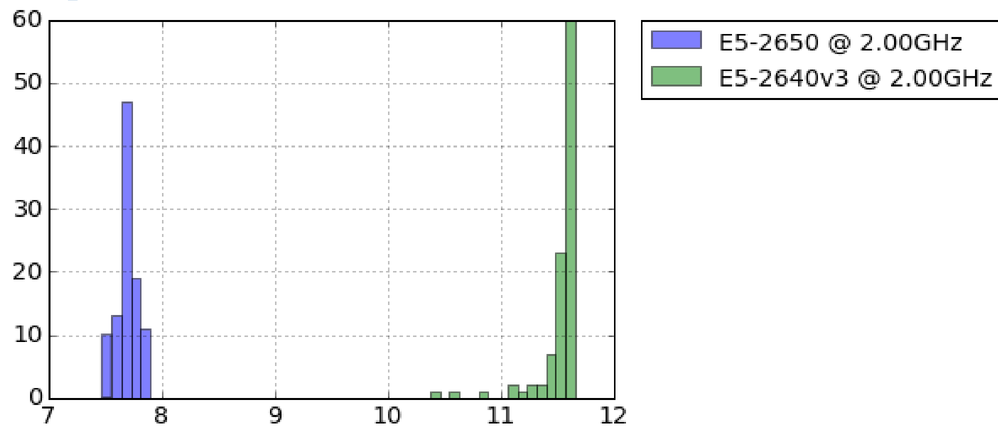
7th February 2017

The problem

The performance of some WLCG applications is much better on worker nodes with Intel Haswell (E5-2600v3) processors than with Sandy Bridge (E5-2600) processors.

Scaled with the HS06 score of the provided job slot, **there is a magic boost of around 45%.** This is much more than the expected improvement in the general purpose performance.

This improvement can be observed with **Dirac benchmark.**



	Sandy Bridge	Haswell
CPU	Dual socket E5-2650 @ 2.00 GHz	Dual socket E5-2640v3 @ 2.60 GHz
RAM	64 GiB RAM DDR3	128 GiB RAM DDR4
OS	SLC 6.8, 2.6.32-642.el6.x86_64	SLC 6.8, 2.6.32-642.el6.x86_64
Compiler	4.4.7	4.4.7
CPU freq	userspace, 2.00 GHz	userspace gov, 2.00 GHz

Approach adopted

Task	Tools
1) Idenfity the major contributors to the runtime	perf, **Intel Software Developer Emulator**
2) Verify that the instruction set used is the same and that there is no	**Intel Software Developer Emulator**
3) Identify peculiarities of the major contributors	perf, source code inspection
4) Try to write synthetic benchmark reproducing the main peculiarities of the workload	C/Python
5) Identify relevant performance counters, draft hypothesis	perf
6) Validate hypothesis on initial workload	

Intel Software Developer Emulator, SDE:

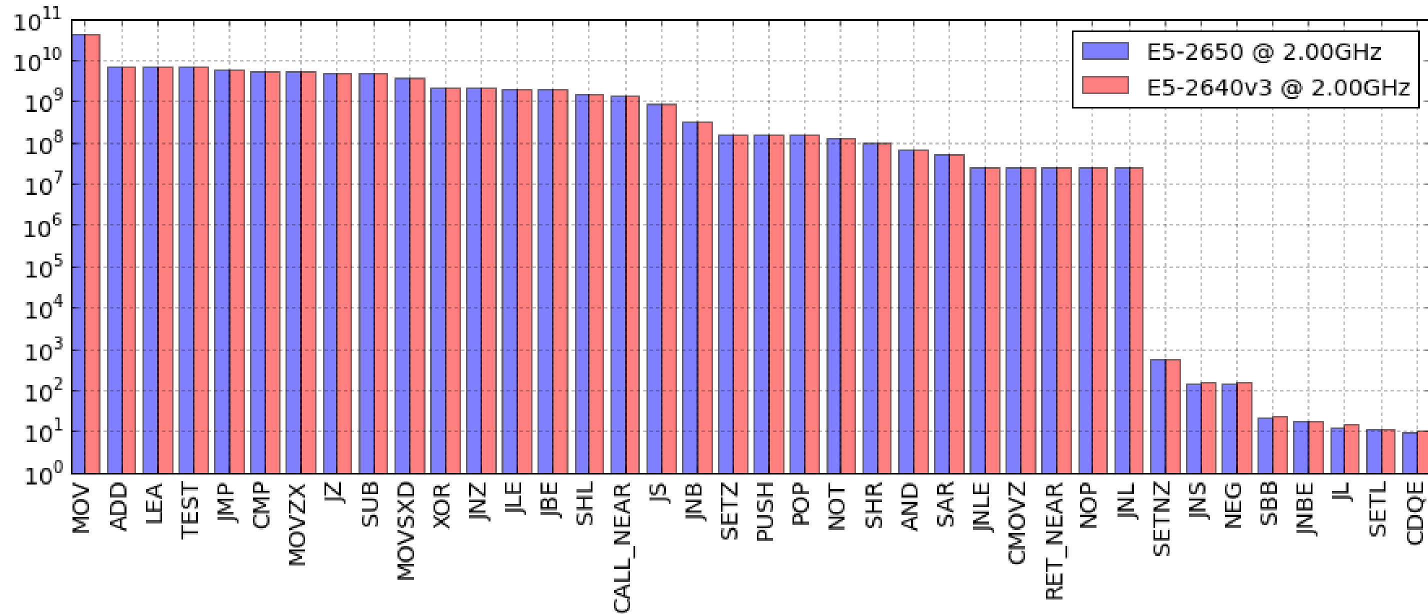
- Uses dynamic binary instrumentation to trace the execution and emulates instructions that are not supported by the architecture according to CPUID (e.g. AVX-512)
- Builds a trace of each instruction executed

Profiling Dirac Benchmark

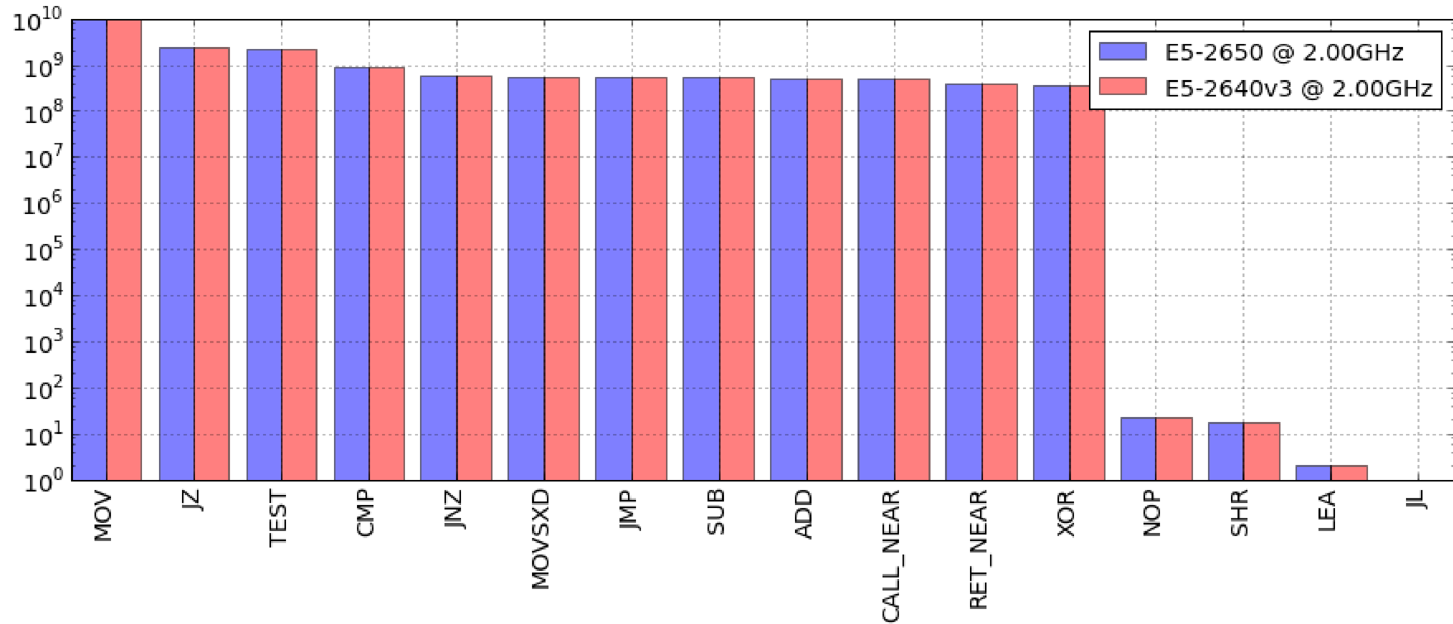
Profile obtained with Intel Software Development Emulator

#rank	total-icount	%	cumulative%	#times-called	function-name	image-name
0:	107791739410	48.671	48.671	25000337	PyEval_EvalFrameEx	/usr/lib64/libpython2.6.so.1.0
1:	19007859626	8.582	57.253	9	PySequence_GetSlice	/usr/lib64/libpython2.6.so.1.0
2:	17182053136	7.758	65.011	542216214	PyFloat_FromDouble	/usr/lib64/libpython2.6.so.1.0
3:	7110203445	3.210	68.222	7	PyDict_Contains	/usr/lib64/libpython2.6.so.1.0
4:	5433499057	2.453	70.675	34221611	__ieee754_log	/lib64/libm.so.6
5:	5338796625	2.411	73.085	1	.text	/usr/lib64/python2.6/lib-dynload/_randommodule.so
6:	4575034173	2.066	75.151	1	PyFrame_Fini	/usr/lib64/libpython2.6.so.1.0
7:	3875789469	1.750	76.901	125024266	PyDict_GetItem	/usr/lib64/libpython2.6.so.1.0
8:	3795513281	1.714	78.615	0	PyFloat_GetMin	/usr/lib64/libpython2.6.so.1.0
9:	3300064436	1.490	80.105	50001091	PyObject_GenericGetAttr	/usr/lib64/libpython2.6.so.1.0
10:	3125018832	1.411	81.516	25000174	PyEval_EvalCodeEx	/usr/lib64/libpython2.6.so.1.0
11:	3075168969	1.389	82.905	150003071	PyType_IsSubtype	/usr/lib64/libpython2.6.so.1.0
12:	2977280263	1.344	84.249	1	initmath	/usr/lib64/python2.6/lib-dynload/mathmodule.so
13:	2650526848	1.197	85.446	50005831	_PyType_Lookup	/usr/lib64/libpython2.6.so.1.0
14:	2438535517	1.101	86.547	143443245	PyNumber_Multiply	/usr/lib64/libpython2.6.so.1.0

Analysis of the main contributors 1/2



Analysis of the main contributors 2/2



PyEval_EvalFrameEx

```
switch (opcode) {
    case NOP:
        <NOP Instruction>
        break
    case LOAD_CONST:
        <LOAD_CONST Instruction>
        break
    case ROT_TWO:
        <ROT_TWO Instruction>
        break
    [...]
    case UNARY_CONVERT:
        <UNARY_CONVERT Instruction>
        break
    [...]
}
```

Large switch statement that constitutes the core of CPython. Dispatches byte-code instructions to corresponding branches.

Synthetic benchmark based on recursive macros, switch argument is incremented at each step (code compiled with –O2)

```
#define CASE(N) \
    case N: \
        flag[N] = N; \
        break;
#define CASE2(N) CASE(N) CASE(N+1)
#define CASE4(N) CASE2(N) CASE2(N+2)
#define CASE8(N) CASE4(N)  CASE4(N+4)
```

Synthetic benchmark results

[root@sandybridge ~]# time numactl --physcpubind=0 --membind=0 ./test

real 0m4.009s

user 0m4.007s

sys 0m0.000s

[root@haswell ~]# time numactl --physcpubind=0 --membind=0 ./test

real 0m0.857s

user 0m0.852s

sys 0m0.005s

Synthetic benchmark results

[root@sandybridge ~]# time numactl --physcpubind=0 --membind=0 ./test

real 0m4.009s

user 0m4.007s

sys 0m0.000s

[root@haswell ~]# time numactl --physcpubind=0 --membind=0 ./test

real 0m0.857s

user 0m0.852s

sys 0m0.005s

More than 4 times slower on SandyBridge

Profile of the synthetic benchmark

SandyBridge

```
    4041.911820      task-clock (msec)
              5      context-switches
              0      cpu-migrations
            297      page-faults
  8,010,609,607      cycles
  5,178,530,946      stalled-cycles-frontend
  4,514,983,977      stalled-cycles-backend
  2,165,529,808      instructions

    808,238,687      branches
    241,795,441      branch-misses

    4.043168304 seconds time elapsed
```

Haswell

```
     866.832373      task-clock (msec)
              3      context-switches
              0      cpu-migrations
            282      page-faults
  1,727,565,963      cycles
<not supported>      stalled-cycles-frontend
<not supported>      stalled-cycles-backend
  2,151,781,261      instructions
    805,581,045      branches
      5,194,391      branch-misses

    0.867114411 seconds time elapsed
```

Profile of the synthetic benchmark

SandyBridge

```
 4041.911820      task-clock (msec)
           5      context-switches
           0      cpu-migrations
         297      page-faults
8,010,609,607     cycles
5,178,530,946     stalled-cycles-frontend
4,514,983,977     stalled-cycles-backend
2,165,529,808     instructions

  808,238,687     branches
  241,795,441     branch-misses

 4.043168304 seconds time elapsed
```

Haswell

```
   866.832373     task-clock (msec)
            3     context-switches
            0     cpu-migrations
          282     page-faults
1,727,565,963     cycles
<not supported>   stalled-cycles-frontend
<not supported>   stalled-cycles-backend
2,151,781,261     instructions
  805,581,045     branches
    5,194,391     branch-misses

  0.867114411 seconds time elapsed
```

30% branches mispredicted on SandyBridge, only 0.6% on Haswell

Mispredicted branches (Synthetic benchmark)

```
       |      Disassembly of section .text:
       |
       |      00000000004004d0 <main>:
       |      main():
       |        sub    $0x8,%rsp
       |        mov    $0x800,%edi
       |        callq  malloc@plt
       |        xor    %edx,%edx
       |  10:   mov    %rdx,%rcx
       |        and    $0x1ff,%ecx
100.00 |        jmpq   fffffffffbffb30
       |        movl   $0x1fe,0x7f8(%rax)
       |        nop
       |  30:   add    $0x1,%edx
       |        cmp    $0x10000000,%edx
```

That jump is actually **jmpq *0x402378(,%rcx,8)**

The switch statement is implemented with a jump table. The argument of the switch is used to access the table, retrieve the address of the corresponding branch, and jump

The same applies to Sandy Bridge and Ivy Bridge

Profiling Dirac benchmark

SandyBridge

```
      65772.689850      task-clock (msec)
                85      context-switches
                 3      cpu-migrations
            77,097      page-faults
   130,297,706,172      cycles
    37,859,268,349      stalled-cycles-frontend
    21,471,656,106      stalled-cycles-backend
   221,916,256,165      instructions

    43,845,979,437      branches
     1,217,252,123      branch-misses

      65.783819363 seconds time elapsed
```

Haswell

```
      43511.147119      task-clock (msec)
               146      context-switches
                 1      cpu-migrations
            77,068      page-faults
    86,767,034,552      cycles
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
   221,821,591,569      instructions
    43,814,386,161      branches
       180,011,844      branch-misses

      43.498133190 seconds time elapsed
```

3% branches mispredicted on SandyBridge, only 0.4% on Haswell

Mispredicted branches (Dirac)

```
      | 4d4:   movl   $0x2,0x58(%rsp)
      |        jmpq   127
      |        nop
      | 4e8:   lea    _PyUnicode_TypeRecords+0xc094,%rsi
      |        mov    %r8d,%eax
      |        movslq (%rsi,%rax,4),%rax
      |        add    %rsi,%rax
97.40 |        jmpq   ffffffca30b2f870
      |        nop
      | 500:   cmpb   $0x7a,(%r14)
      |        je     366
      |        mov    _PyImport_DynLoadFiletab+0x318,%rax
      |        mov    (%rax),%eax
      |        mov    %eax,(%r8)
      |        addl   $0x1,0x80(%r12)
```

Large majority of mispredicted branches happens in PyEval_EvalFrameEx on the switch statement, which is again an indirect jump.

The jmp correctly decoded would be **jmpq *%rax**.

Conclusions

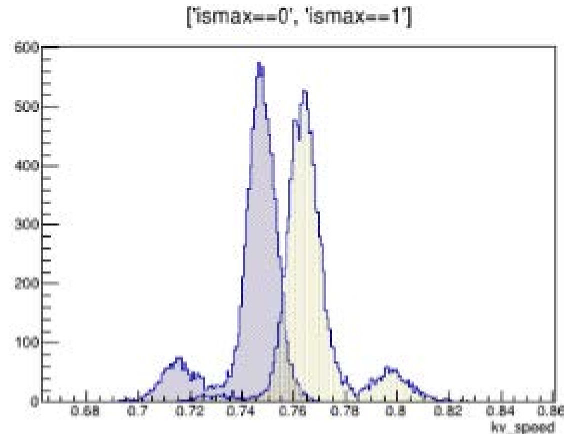
- The major contributor to the Dirac benchmark is **PyEval_EvalFrameEx, which is the core of CPython interpreter.**
- Switch/case statement is the basic building block of this routine, and it heavily benefits from a performant **branch prediction unit to correctly identify the target of the indirect jump**
- This type of workload is probably very relevant also for other virtual machines implementations and particular attention was given to the Branch Prediction when designing the new Haswell microarchitecture

Investigation of the dual peak effect in ATLAS Kit Validation

Marco Guerri

7th February 2017

The problem

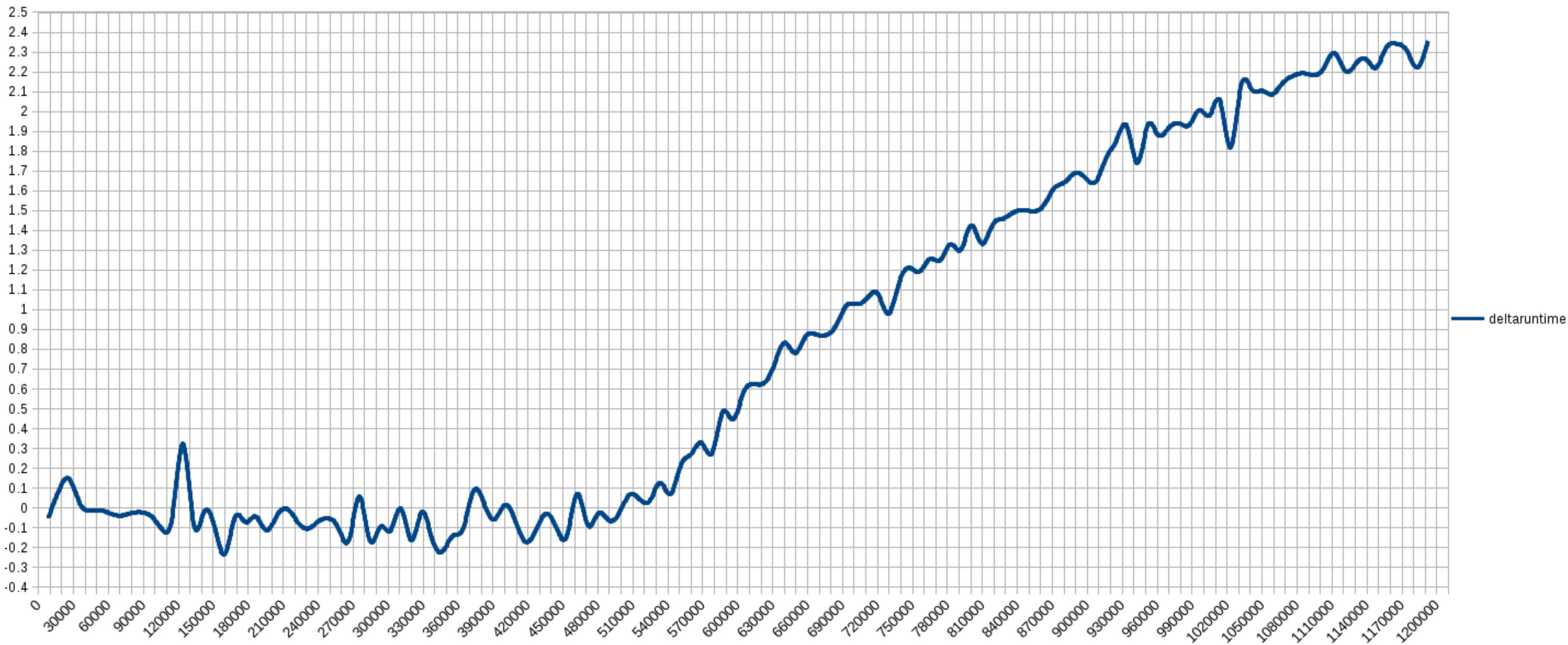


The distribution of ATLAS Kit Validation runs has a faster mode and a slower one.

Slower runs seem to be always associated to CPU1, in particular, KV runs slower on hyperthread 8 and 24, 2 threads which belong to the first physical core of the second processor.

```
       Disassembly of section .text:

       0000000000064a90 <__sin_avx>:
       __sin_avx():
 5.46     push   %r12
24.40     vmovap %xmm0,%xmm2
 3.26     push   %rbp
 1.74     push   %rbx
          sub    $0x70,%rsp
 0.62     vstmxc 0x50(%rsp)
 1.81     mov    0x50(%rsp),%ebx
          mov    %ebx,%eax
 0.60     and    $0x9f,%ah
 0.73     cmp    %eax,%ebx
          mov    %eax,0x60(%rsp)
 0.34   ↓ jne    1dfb
          xor    %ebp,%ebp
     29:  vmovsd %xmm2,(%rsp)
 0.16     mov    (%rsp),%rdx
 0.23     mov    %rdx,%rcx
          sar    $0x20,%rcx
          mov    %ecx,%eax
          and    $0x7fffffff,%eax
 0.39     cmp    $0x3e4fffff,%eax
        ↓ jle    e0
          cmp    $0x3fcfffff,%eax
 0.08   ↓ jg     100
```


2.5
2.4
2.3
2.2
2.1
2
1.9
1.8
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0
30000
60000
90000
120000
150000
180000
210000
240000
270000
300000
330000
360000
390000
420000
450000
480000
510000
540000
570000
600000
630000
660000
690000
720000
750000
780000
810000
840000
870000
900000
930000
960000
990000
1020000
1050000
1080000
1110000
1140000
1170000
1200000
deltaruntime