

Profiling CPU-bound workloads on Intel Haswell-EP platforms

M Guerri¹, D Giordano¹ and C Cordeiro¹

¹ CERN

E-mail: marco.guerri@cern.ch

Abstract. With the increasing adoption of public and private cloud resources to support the demands in terms of computing capacity of the WLCG, the HEP community has begun studying several benchmarking applications aimed at continuously assessing the performance of virtual machines procured from commercial providers. In order to characterise the behaviour of these benchmarks, in-depth profiling activities have been carried out. In this document we outline our experience in profiling one specific application, the ATLAS Kit Validation, in an attempt to explain an unexpected distribution in the performance samples obtained on systems based on Intel Haswell-EP processors.

1. Introduction

Nowadays the majority of the experiments' workloads running on the WLCG are simulation jobs, where most of the computing time is spent in CPU-bound activities. For this reason the adoption of fast benchmarks representative of these applications can be beneficial for monitoring and resource allocation purposes, providing real-time information on the performance delivered by virtual machines in the cloud or job slots in a Grid site. A number of fast benchmarks are being studied by the HEP community. Within the HEPiX Benchmarking Working Group [1], two fast benchmarks are in particular under deep investigation: Dirac Benchmark 2012 (DB12) [2] and ATLAS Kit Validation (KV) [3].

2. KV Benchmark

KV is the toolkit adopted by the ATLAS collaboration for the validation of their software installation in Grid sites. The tests include, among others, the GEANT4 [4] simulation of the ATLAS detector, which has been repurposed to benchmark compute resources against CPU-bound workloads. The KV benchmark simulates n independent events consisting of a single muon particle propagating through the ATLAS detector. The CPU time needed to elaborate each event is recorded and the average over the n events is computed as the final benchmark result. By excluding the first event in the sequence from the final average, spurious effects such as the overhead coming from the initialisation of the software libraries and the configuration of the simulation parameters (detector geometry, list of particles, properties of the materials) are disregarded from the measurement.

The KV benchmark is single threaded as are most of the LHC experiments' software applications. In order to fully utilise compute resources and reproduce the worst case load scenario, where all compute slots are running jobs and CPU idle time is minimised, the benchmark by default is configured to run in parallel with as many threads as the number of logical cores provided by the server and the benchmark result is calculated as the arithmetic average over all threads.

In the past few years the KV benchmark has been used to assess the performance of resources provided by commercial cloud providers and Grid sites. The results have been compared with the average CPU time required to simulate ATLAS events, as reported by the experiment's job monitoring framework. Details about this activity and the correlation among benchmark results and job duration are reported in [5].

3. Investigating KV results

One of the several environments benchmarked using KV consisted of 240 Haswell-EP hypervisors providing compute resources as VMs dedicated to the CERN Tier-0 batch system. The servers were fitted with two 8-core Intel Xeon E5-2630v3 processors, with a total number of 32 threads per server with simultaneous multithreading enabled, and 64 GiB of DDR4 RAM in fully balanced configuration, with each memory channel populated with the same number of DIMMs of equal capacity. All the hypervisors were configured to host 2 VMs installed with SLC6 providing 16 virtual processing units each, with pairs of VMs confined to separate NUMA domains. KV was executed with 16 parallel processes on each VM, resulting in the full utilisation of the hardware threads provided by the machine, and the final aggregated result was obtained as the average over the 16 simulations. The samples collected from the VMs under test resulted in a bimodal gaussian distribution, with a 2% difference between the mean of the two modes as shown in Figure 1.

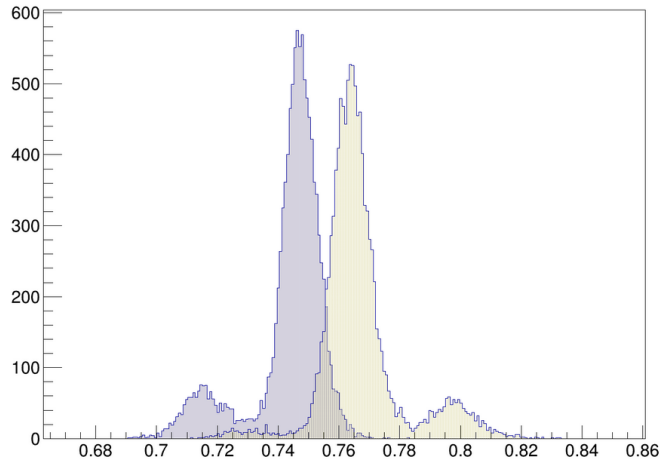


Figure 1: Distribution of the average KV execution speed [sec/evt] on VMs with 16 virtual CPUs.

In order to ease the performance analysis, the virtualisation layer was temporarily removed and simultaneous multithreading disabled, running KV on the bare-metal server installed with CentOS 7 while setting scheduling affinity sequentially to each physical core. The results shown in table 1 highlighted a 12% difference in the average simulation time when binding KV to core 8, the first core of the second processor (CPU1).

Table 1: Average simulation time (sec/evt) on bare-metal server running CentOS 7

CPU0		CPU1	
Core 0	0.80	Core 8	0.90
Core 1	0.80	Core 9	0.81
Core 2	0.80	Core 10	0.79
Core 3	0.81	Core 11	0.80
Core 4	0.81	Core 12	0.80
Core 5	0.81	Core 13	0.81
Core 6	0.80	Core 14	0.79
Core 7	0.81	Core 15	0.81

Further tests confirmed the same behaviour also with simultaneous multithreading enabled, with slower performance on threads 8 and 24, both belonging to the first core of CPU1. Single-threaded runs in a virtualised environment showed a 16% increase in the final average simulation time when running on virtual processors corresponding to hardware threads 8 and 24, confirming the 2% spread that stood out from the statistical distribution after averaging the results over all 8 virtual processors within each VM. This performance discrepancy was not observed with any other benchmark under study within the HEPiX Benchmarking Working Group, namely DB12 and Whetstone.

4. Profiling single events

A deeper dive in the logs showed that all the events required more time to be simulated on core 8. In particular, event 73 was identified as the one requiring the longest simulation time: 34 seconds on core 8 against 29 seconds on any other core, making it a good candidate for a detailed analysis. Profiling a single event required a way to attach *perf* to KV only during the time frame where one specific simulation was being executed. The solution adopted consisted of a Python script which would monitor KV log files via *pyinotify* to detect the beginning of event 73 and then attach *perf* to the process until the detection of the end of the event. One essential requirement for this to work correctly was to disable all the buffering carried out at any

layer of the stack before data actually reached the storage device. KV writes log files via Python code, therefore the following points had to be taken into consideration:

- A call to the *open* function in Python results in the invocation of libc *fopen*. Python can modify libc buffering behaviour via the libc function *setvbuf* if *buffering* parameter is specified, otherwise the default buffering mode is used.
- The environment variable *PYTHONUNBUFFERED* can be used to turn off buffering of *stdin*, *stdout* and *stderr*.

Without control over the Python source code, unbuffered I/O was implemented by invoking KV with *stdbuf -o0* command and redirecting *stdout* and *stderr* to an additional file that would be monitored by the *pyinotify* script. This approach works correctly even when commands are piped together. The profile of the execution was obtained with *perf record*. Results for core 0 and core 8 are shown respectively in Listing 1 and Listing 2, limiting the output to functions that contribute for more than 1% of the runtime.

# Samples: 125K of event 'cycles:pp'				
# Event count (approx.): 99035639955				
#				
#	Overhead	Command	Shared Object	Symbol
#				
	12.72%	athena.py	libm-2.17.so	[.] __ieee754_log_avx
	7.36%	athena.py	libm-2.17.so	[.] __ieee754_exp_avx
	5.51%	athena.py	libG4global.so	[.] G4PhysicsVector::Value
	3.64%	athena.py	libG4processes.so	[.] G4UniversalFluctuation::SampleFluctuations
	3.25%	athena.py	libm-2.17.so	[.] __GI__exp
	2.67%	athena.py	libm-2.17.so	[.] __sin_avx
	2.54%	athena.py	libMagFieldServices.so	[.] BFieldCache::getB
	2.26%	athena.py	libG4tracking.so	[.] G4SteppingManager::DefinePhysicalStepLength
	2.13%	athena.py	libm-2.17.so	[.] __cos_avx
	1.81%	athena.py	libG4geometry.so	[.] G4Navigator::LocateGlobalPointWithinVolume
	1.69%	athena.py	libG4tracking.so	[.] G4SteppingManager::InvokePSDIP
	1.62%	athena.py	libm-2.17.so	[.] __log10_finite
	1.46%	athena.py	libm-2.17.so	[.] __sincos
	1.45%	athena.py	libG4geometry.so	[.] G4PropagatorInField::ComputeStep
	1.28%	athena.py	libG4processes.so	[.] G4UrbanMscModel93::SampleCosineTheta
	1.26%	athena.py	libG4tracking.so	[.] G4SteppingManager::Stepping
	1.17%	athena.py	libG4geometry.so	[.] G4AtlasRK4::Stepper
	1.14%	athena.py	libG4processes.so	[.] G4VEnergyLossProcess::PostStepGetPhysicalInter
	1.06%	athena.py	libG4processes.so	[.] G4ElectroNuclearCrossSection::GetZandACrossSec

Listing 1: Recording of event 73 on core 0.

# Samples: 141K of event 'cycles:pp'				
# Event count (approx.): 111860905419				
#				
#	Overhead	Command	Shared Object	Symbol
#				
	11.38%	athena.py	libm-2.17.so	[.] __ieee754_log_avx
	6.44%	athena.py	libm-2.17.so	[.] __ieee754_exp_avx
	4.90%	athena.py	libG4global.so	[.] G4PhysicsVector::Value
	3.38%	athena.py	libG4processes.so	[.] G4UniversalFluctuation::SampleFluctuations
	2.88%	athena.py	libm-2.17.so	[.] __GI__exp
	2.64%	athena.py	libm-2.17.so	[.] __sin_avx
	2.51%	athena.py	libMagFieldServices.so	[.] BFieldCache::getB
	1.93%	athena.py	libG4tracking.so	[.] G4SteppingManager::DefinePhysicalStepLength
	1.83%	athena.py	libm-2.17.so	[.] __cos_avx
	1.78%	athena.py	libG4geometry.so	[.] G4Navigator::LocateGlobalPointWithinVolume
	1.59%	athena.py	libm-2.17.so	[.] __log10_finite
	1.45%	athena.py	libG4tracking.so	[.] G4SteppingManager::InvokePSDIP
	1.37%	athena.py	libG4geometry.so	[.] G4PropagatorInField::ComputeStep
	1.36%	athena.py	libm-2.17.so	[.] __sincos
	1.28%	athena.py	libG4processes.so	[.] G4VEnergyLossProcess::
	→	PostStepGetPhysicalInteractionLength		
	1.18%	athena.py	libG4tracking.so	[.] G4SteppingManager::Stepping
	1.15%	athena.py	libG4processes.so	[.] G4UrbanMscModel93::SampleCosineTheta
	1.01%	athena.py	libG4tracking.so	[.] G4SteppingManager::InvokeAlongStepDoItProcs
	1.01%	athena.py	libG4geometry.so	[.] G4AtlasRK4::Stepper

Listing 2: Recording of event 73 on core 8.

Excluding the higher number of events (unhalted core cycles) on core 8, which was expected due to the longer runtime, it was not possible to identify any major differences between the two runs. A list of

the top contributors to the additional clock cycles required when running on core 8 was extracted from the recordings. Results are shown in Table 2.

G4StepLimiter::PostStepGetPhysicalInteractionLength	2.67%
__sin_avx	2.45%
G4VEnergyLossProcess::PostStepGetPhysicalInteractionLength	2.37%
FADS::FadsSteppingAction::UserSteppingAction	2.33%
_ZN12G4FieldTrackC1ERKN5CLHEP10Hep3VectorES3_dddPS2_@plt	2.27%
BFieldCache::getB	2.22%
_ZN11TrackHelperC1EPK7G4Track@plt	2.08%
G4CrossSectionDataStore::GetCrossSection	2.06%
G4SteppingManager::InvokeAlongStepDoItProcs	2.01%
G4Transportation::AlongStepGetPhysicalInteractionLength	1.85%
G4FieldTrack::G4FieldTrack	1.69%
std::_Rb_tree_increment	1.66%
G4Mag_EqRhs::SetChargeMomentumMass	1.63%
G4Navigator::LocateGlobalPointWithinVolume	1.56%
G4ParticleChangeForTransport::UpdateStepForAlongStep	1.43%
_log10_finite	1.39%

Table 2: Major contributors to the additional runtime on core 8 (percentage of the additional clock cycles)

5. Profiling the major contributors

The functions appearing in Table 2 were profiled to obtain a detailed breakdown of the weight of each assembly instruction in terms of clock cycles. The most relevant recordings are reported in Appendix 8. The results highlighted an interesting pattern on most of the major contributors to the additional runtime on core 8: it appeared that an unexpectedly large number of cycles was spent on the initial instructions of each function, most of which could not justify such a high cost. In some cases a *mov* of a literal value to a register was attributed 7.5% of the samples collected for that specific function (Listing 9). Such behaviour was common to all cores and could not explain the performance asymmetry highlighted by the benchmark, but it was reasonable to hypothesize a correlation with this peculiar profile. A first hypothesis pointed in the direction of instruction cache misses as the reason behind the overhead appearing on the initial instructions. ATLAS Kit Validation seemed a good candidate for a large number of such events, with its code segment consisting of more than 300 shared libraries.

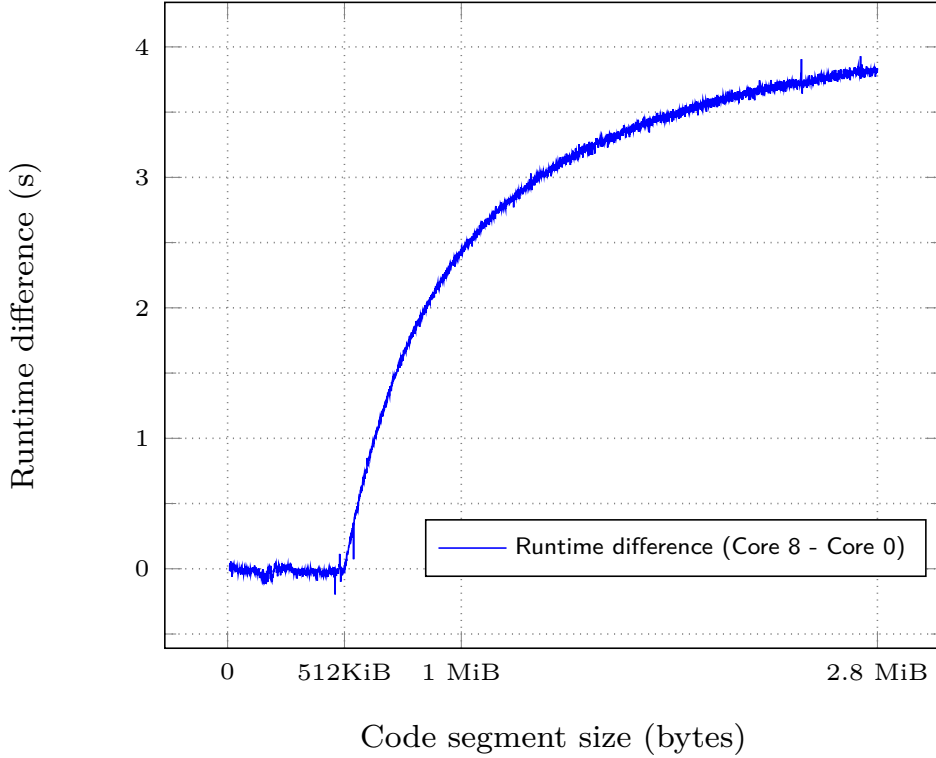
6. A synthetic benchmark to generate instruction cache misses

A synthetic benchmark aimed at causing a large number of instruction cache misses was automatically generated using a Python script [6]. The resulting C code was compiled under CentOS 7 with gcc 4.8.5 and optimizations disabled (*-O0*). The benchmark consists of a large number of dummy functions which are called randomly at runtime to generate repeated instruction cache misses in L1i and L2. Interestingly, the results highlighted the same performance asymmetry observed with ATLAS Kit Validation, with a 20% increase in runtime on core 8 with respect to the other cores. This effect could only be observed on Haswell-EP, Haswell-EX (quad-socket, with a single core on two different sockets affected) and Broadwell-EP systems. It was not possible to reproduce a similar performance asymmetry on Sandy Bridge and Ivy Bridge platforms. Further investigations succeeded in generating workloads with similar relative number of L1i and L2 instruction cache misses on Haswell and Broadwell, but without any discrepancy in execution time, disproving the initial hypothesis and making it necessary to seek for an alternative explanation.

7. An additional benchmark with customizable code segment size

A second synthetic benchmark was written in order to verify the existence of a correlation between the size of the code segment and the slowdown observed on core 8 [7]. This benchmark also aims to generate instruction cache misses, but within a code segment whose size can be controlled via a command line

Figure 2: Runtime difference between core 8 and core 0



argument. The code makes use of a large switch statement that dispatches randomly generated opcodes in loop. By feeding the switch with *opcode mod n*, the number of case branches effectively used at runtime is limited to n . The higher n , the larger the number of case branches that can be reached. By sampling the maximum and minimum values of *rip* register, a reliable estimate of the size of the hot portion of the code segment is obtained. The benchmark takes n as command line argument and prints a tuple consisting of three values: *size code segment*, *runtime without initialization* and n . The graph in Figure 2 shows the difference in runtime between core 8 and core 0 with respect to the size of the code segment. After having filtered out a negligible number of outliers (1.5% of the measurements), the samples collected clearly highlighted a performance degradation beyond approximately 512 KiB, with core 8 running up to 4 seconds slower (9.4%) in the 3 MB area. 512 KiB is an interesting number, as it corresponds to the space addressable with 128 iTLB entries using 4 KiB pages: this matches exactly the size of the iTLB on Haswell and Broadwell architectures. With a code segment larger than 512 KiB, a heavily nonlinear execution flow starts generating an increasingly large number of iTLB misses and a diverging performance profile possibly points in the direction of a different behaviour of the pipeline front-end in handling such events. The test was repeated by making use of the huge pages support of the operating system (2MiB) to map the code segment of the benchmark. With this setup no performance difference was observed between core 8 and the other cores on the system.

8. Conclusions

When running specific workloads on dual socket Haswell-EP platforms, we observed a difference in performance between the first core of the second CPU and the other cores on the system. Synthetic benchmarks written during this study require up to 20% additional time to complete on the first core of the second CPU. This asymmetry has been reproduced on Haswell-EP, Haswell-EX and Broadwell-EP platforms. The results of our deep investigation suggest that lower performance on core 8 be caused by a different behaviour in handling iTLB misses: as a consequence, different runtime can be observed only with workloads that generate a high number of such events. At this stage we are unable to provide a conclusive explanation for this asymmetry.

Appendices

A. Function annotations

Percent	Source code & Disassembly of libG4processes.so for cycles:pp

	Disassembly of section .text:
	00000000032fab0 <G4StepLimiter::PostStepGetPhysicalInteractionLength(G4Track const&, double, G4ForceCondition*)>:
	_ZN13G4StepLimiter36PostStepGetPhysicalInteractionLengthERK7G4TrackdP16G4ForceCondition():
39.78	32fab0: push %rbx
0.12	32fab1: movl \$0x2, (%rdx)
	_ZNK7G4Track9GetVolumeEv():
0.00	32fab7: xor %eax, %eax
	_ZNK24G4ReferenceCountedHandleI12G4VTouchableEcvbEv():
0.73	32fab9: mov 0x40(%rsi), %rdx
	_ZN13G4StepLimiter36PostStepGetPhysicalInteractionLengthERK7G4TrackdP16G4ForceCondition():
0.24	32fabd: mov %rsi, %rbx
	_ZNK7G4Track9GetVolumeEv():
2.57	32fac0: test %rdx, %rdx
0.00	32fac3: je 32fad1 <G4StepLimiter::PostStepGetPhysicalInteractionLength(G4Track const&, double, ...)+0x21>
	_ZNK24G4ReferenceCountedHandleI12G4VTouchableEptEv():
5.88	32fac5: mov 0x8(%rdx), %rdi
	_ZNK7G4Track9GetVolumeEv():
0.24	32fac9: xor %esi, %esi
7.47	32facb: mov (%rdi), %rax
4.04	32face: callq *0x20(%rax)
9.06	32fad1: mov 0x28(%rax), %rax
10.04	32fad5: mov 0x40(%rax), %rdi
	_ZNK15G4LogicalVolume13GetUserLimitsEv():
0.00	32fad9: test %rdi, %rdi
0.61	32fadc: je 32faf8 <G4StepLimiter::PostStepGetPhysicalInteractionLength(G4Track const&, double, ...)+0x48>
	[...]

Listing 3: G4StepLimiter::PostStepGetPhysicalInteractionLength cycles annotation

Percent	Source code & Disassembly of libm-2.17.so for cycles:pp

	Disassembly of section .text:
	0000000000064a30 <__sin_avx>:
	__sin_avx():
6.73	64a30: push %r12
24.77	64a32: vmovapd %xmm0, %xmm2
4.10	64a36: push %rbp
1.23	64a37: push %rbx
0.00	64a38: sub \$0x70, %rsp
0.70	64a3c: vstmxcsr 0x50(%rsp)
2.01	64a42: mov 0x50(%rsp), %ebx
0.00	64a46: mov %ebx, %eax
0.67	64a48: and \$0x9f, %ah
0.59	64a4b: cmp %eax, %ebx
0.00	64a4d: mov %eax, 0x60(%rsp)
0.32	64a51: jne 6682b <__sin_avx+0x1dfb>
0.00	64a57: xor %ebp, %ebp
0.00	64a59: vmovsd %xmm2, (%rsp)
0.03	64a5e: mov (%rsp), %rdx
	[...]

Listing 4: __sin_avx cycles annotation

Percent	Source code & Disassembly of libG4processes.so for cycles:pp

	Disassembly of section .text:
	00000000003c77b0 <G4VEnergyLossProcess::PostStepGetPhysicalInteractionLength(G4Track const&, double, G4ForceCondition*)>:
	_ZN20G4VEnergyLossProcess36PostStepGetPhysicalInteractionLengthERK7G4TrackdP16G4ForceCondition():
15.07	3c77b0: push %r14
0.00	3c77b2: movapd %xmm0, %xmm3
0.17	3c77b6: push %r13
0.11	3c77b8: push %r12
0.28	3c77ba: push %rbp
0.00	3c77bb: push %rbx
0.11	3c77bc: mov %rdi, %rbx
0.50	3c77bf: sub \$0x50, %rsp
	_ZNK7G4Track21GetMaterialCutsCoupleEv():
0.17	3c77c3: mov 0x78(%rsi), %rax
	_ZN20G4VEnergyLossProcess36PostStepGetPhysicalInteractionLengthERK7G4TrackdP16G4ForceCondition():
0.06	3c77c7: movl \$0x2, (%rdx)
	_ZNK7G4Track21GetMaterialCutsCoupleEv():
0.94	3c77cd: mov 0x10(%rax), %rax
1.32	3c77d1: mov 0x68(%rax), %rax
	_ZN20G4VEnergyLossProcess14DefineMaterialEPK20G4MaterialCutsCouple():
0.50	3c77d5: cmp 0x3c0(%rdi), %rax
1.16	3c77dc: je 3c780c <G4VEnergyLossProcess::PostStepGetPhysicalInteractionLength(...)+0x5c>
1.82	3c77de: mov 0x10(%rax), %rdx
	[...]

Listing 5: G4VEnergyLossProcess::PostStepGetPhysicalInteractionLength cycles annotation

Percent	Source code & Disassembly of libFadsActions.so for cycles:pp

:	Disassembly of section .text:
:	:
:	0000000000087f0 <FADS::FadsSteppingAction::UserSteppingAction(G4Step const*)>:
:	_ZN4FADS18FadsSteppingAction18UserSteppingActionEPK6G4Step():
28.47 :	87f0: push %r12
0.00 :	87f2: push %rbp
0.10 :	87f3: mov %rdi,%rbp
0.10 :	87f6: push %rbx
0.19 :	87f7: mov %rsi,%rbx
0.00 :	87fa: sub \$0x20,%rsp
8.30 :	87fe: cmpb \$0x0,0x8fa3(%rip)
0.58 :	8805: je 8900 <FADS::FadsSteppingAction::UserSteppingAction(G4Step const*)+0x110>
2.99 :	880b: cmpb \$0x0,0x8d8e(%rip) # 115a0 <FADS::FadsSteppingAction::UserSteppingAction(G4Step const*)::first>
0.97 :	8812: jne 8960 <FADS::FadsSteppingAction::UserSteppingAction(G4Step const*)+0x170>
0.68 :	8818: mov 0x8f91(%rip),%rax # 117b0 <FADS::FadsSteppingAction::UserSteppingAction(G4Step const*)::act>
11.87 :	881f: mov -0x18(%rax),%rsi
0.00 :	8823: test %rsi,%rsi
0.87 :	8826: je 88a5 <FADS::FadsSteppingAction::UserSteppingAction(G4Step const*)+0xb5>
0.00 :	8828: mov 0x8891(%rip),%r12 # 110c0 <_DYNAMIC+0x388>
[...]	

Listing 6: FADS::FadsSteppingAction::UserSteppingAction cycles annotation

Percent	Source code & Disassembly of libG4processes.so for cycles:pp

:	Disassembly of section .text:
:	:
:	0000000005bbc00 <G4CrossSectionDataStore::GetCrossSection(G4DynamicParticle const*, G4Element const*, double)>:
:	_ZN23G4CrossSectionDataStore15GetCrossSectionEPK17G4DynamicParticlePK9G4Elementd():
14.85 :	5bbc00: push %r15
0.17 :	5bbc02: push %r14
1.26 :	5bbc04: push %r13
1.76 :	5bbc06: mov %rdx,%r13
0.25 :	5bbc09: push %r12
0.00 :	5bbc0b: mov %rsi,%r12
1.17 :	5bbc0e: push %rbp
0.84 :	5bbc0f: mov %rdi,%rbp
0.84 :	5bbc12: push %rbx
1.01 :	5bbc13: sub \$0x58,%rsp
5.87 :	5bbc17: mov 0x18(%rdi),%eax
0.00 :	5bbc1a: movsd %xmm0,0x8(%rsp)
0.25 :	5bbc20: test %eax,%eax
0.00 :	5bbc22: je 5bbf6c <G4CrossSectionDataStore::GetCrossSection(...)+0x59>
0.84 :	5bbc28: sub \$0x1,%eax
[...]	

Listing 7: G4CrossSectionDataStore::GetCrossSection cycles annotation

Percent	Source code & Disassembly of libG4geometry.so for cycles:pp

:	Disassembly of section .text:
:	:
:	0000000000bc9c0 <G4FieldTrack::G4FieldTrack(...)>:
:	_ZN12G4FieldTrackC2ERKN5CLHEP10Hep3VectorES3_dddPS2_():
7.76 :	bc9c0: sub \$0x88,%rsp
0.13 :	bc9c7: movsd %xmm2,0x40(%rdi)
0.92 :	bc9cc: addsd %xmm2,%xmm2
0.39 :	bc9d0: movsd %xmm4,0x48(%rdi)
0.00 :	bc9d5: movabs \$0x7fffffff,%rax
0.26 :	bc9df: movq \$0x0,0x58(%rdi)
0.66 :	bc9e7: movapd %xmm0,%xmm3
1.18 :	bc9eb: movsd %xmm1,0x38(%rdi)
0.00 :	bc9f0: movsd %xmm5,0x50(%rdi)
0.00 :	bc9f5: movq \$0x0,0x60(%rdi)
1.58 :	bc9fd: addsd %xmm1,%xmm2
0.26 :	bca01: movq \$0x0,0x68(%rdi)
0.53 :	bca09: movq \$0x0,0x70(%rdi)
0.13 :	bca11: movq \$0x0,0x78(%rdi)
1.18 :	bca19: movq \$0x0,0x80(%rdi)

Listing 8: G4FieldTrack::G4FieldTrack cycles annotation

Percent	Source code & Disassembly of libm-2.17.so for cycles:pp

:	Disassembly of section .text:
:	:
:	000000000014690 <__log10_finite>:
:	__ieee754_log10():
7.59 :	14690: movabs \$0xffffffff,%rcx
0.00 :	1469a: movq %xmm0,%rax
0.00 :	1469f: cmp %rcx,%rax
1.60 :	146a2: jg 146d8 <__log10_finite+0x48>
0.00 :	146a4: movabs \$0xffffffff,%rdx
0.00 :	146ae: test %rdx,%rax
0.00 :	146b1: je 14770 <__log10_finite+0xe0>
0.00 :	146b7: test %rax,%rax
0.00 :	146ba: js 1478f <__log10_finite+0xff>
0.00 :	146c0: mulsd 0x74b18(%rip),%xmm0 # 891e0 <sqrt_2.2977+0x10>
0.00 :	146c8: mov \$0xffffffff,%ecx
0.00 :	146cd: movq %xmm0,%rdx
0.00 :	146d2: jmp 146e0 <__log10_finite+0x50>
0.00 :	146d4: nopl 0x0(%rax)
0.27 :	146d8: mov %rax,%rdx

Listing 9: __log10.finite cycles annotation

References

- [1] Hepix benchmarking working group. www.hepix.org/e10227/e10327/e10325.
- [2] Dirac benchmark 2012. gitlab.cern.ch/mcnab/dirac-benchmark/tree/master.
- [3] Alessandro De Salvo and Franco Brasolin. Benchmarking the atlas software through the kit validation engine. *Journal of Physics: Conference Series*, 219(4):042037, 2010.
- [4] G Cosmo and the Geant4 Collaboration. Geant4 - towards major release 10. *Journal of Physics: Conference Series*, 513(2):022005, 2014.
- [5] D Giordano et al. Benchmarking cloud resources for hep. *to appear in J. Phys.: Conf. Ser.*, 2016.
- [6] Synthetic benchmark which generates instruction cache misses. <https://gitlab.cern.ch/snippets/216>,
(mirror: <https://gist.github.com/marcoguerri/eac831af953dbdd215687c270218d0bd>).
- [7] Synthetic benchmark which generates instruction cache misses within a code segment of adjustable size. <https://gitlab.cern.ch/snippets/217>
(mirror: <https://gist.github.com/marcoguerri/9cd145bc2f3cd8917f0ddab95406158f>).