Core-Level Performance Engineering with the Open-Source Architecture Code Analyzer (OSACA) and the Compiler Explorer

Jan Laukemann
Georg Hager
jan.laukemann@fau.de
georg.hager@fau.de
Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany

ABSTRACT

While many developers put a lot of effort into optimizing large-scale parallelism, they often neglect the importance of an efficient serial code. Even worse, slow serial code tends to scale very well, hiding the fact that resources are wasted because no definite hardware performance limit ("bottleneck") is exhausted. This tutorial conveys the required knowledge to develop a thorough understanding of the interactions between software and hardware on the level of a single CPU core and the lowest memory hierarchy level (the L1 cache). We introduce general out-of-order core architectures and their typical performance bottlenecks using modern x86-64 (Intel Ice Lake) and ARM (Fujitsu A64FX) processors as examples. We then go into detail about x86 and AArch64 assembly code, specifically including vectorization (SIMD), pipeline utilization, critical paths, and loop-carried dependencies. We also demonstrate performance analysis and performance engineering using the Open-Source Architecture Code Analyzer (OSACA) in combination with a dedicated instance of the well-known Compiler Explorer. Various hands-on exercises allow attendees to make their own experiments and measurements and identify in-core performance bottlenecks. Furthermore, we show real-life use cases to emphasize how profitable in-core performance engineering can be.

CCS CONCEPTS

• **General and reference → Performance**; • **Computing methodologies → Massively parallel and high-performance simulations**.

KEYWORDS

performance modeling, performance engineering, in-core

ACM Reference Format:
Jan Laukemann and Georg Hager. 2023. Core-Level Performance Engineering with the Open-Source Architecture Code Analyzer (OSACA) and the Compiler Explorer. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3583716

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3583716

1 INTRODUCTION

1.1 Model-based performance engineering

Understanding an application and its underlying code is crucial for performance-aware software development in scientific computing and high performance computing (HPC). Performance models of such applications can provide benefits in various aspects, such as energy efficiency, architectural exploration, and performance optimization. The scientific literature nowadays differentiates between two different approaches to performance modeling [6]: *White-box* (or *first-principles*) models which require prerequisite knowledge about the hardware and its interaction with the software (such as vendor documentation), and *black-box* models which need no previous information about the system, e.g., an AI model trained via machine learning on empirical data. While black-box models can be automated and are faster to set up, they often lack accuracy due to the intricacies of a complex system like a CPU and its interactions on numerous levels with the code and other hardware. On the other hand, a white-box model requires more work through reverse engineering or studying vendor documentation, but it allows deeper insight into hardware-software interaction. By its nature, the model is explicit about the reasons for its predictions and thus often provides a beneficial learning experience for the user. Combined with some empirical data like measurement-based assumptions, a white-box model can become a *gray-box* model and can be refined to achieve higher accuracy. One example for this is the well-known Roofline model [17]. It is based on the (first-principle) assumption that the performance is either limited by the peak floating-point (FP) performance or the memory bandwidth of a CPU; both can be obtained from the system specifications but can be refined using empirical data, taking into account the fact that, in practice, the sustained memory bandwidth is significantly lower than its theoretical value.

This tutorial covers performance engineering on the CPU core level based on white-box models of code execution. The core level should be the starting point of any optimization process since a good single-core performance is crucial for fully utilizing a system's resources. The in-core prediction of code execution time on a modern CPU considers the execution of instructions with all the required data residing in registers and the L1 cache. We can

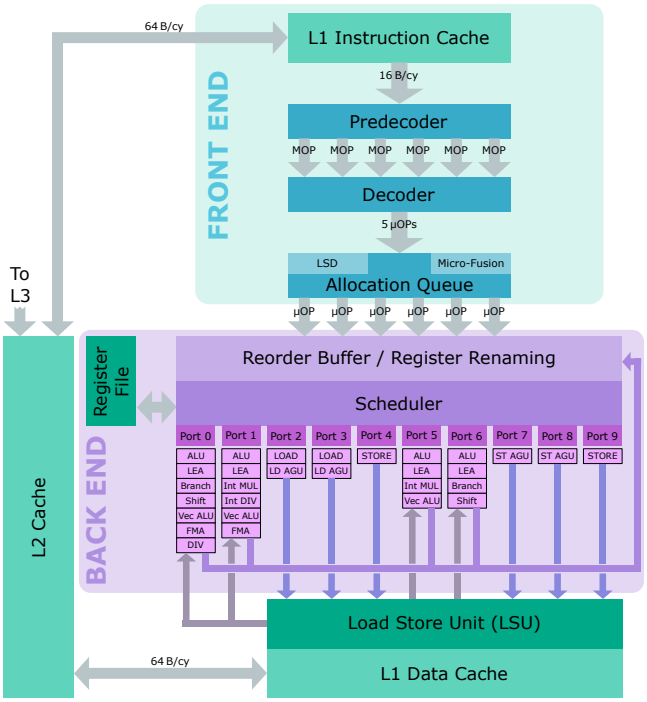


Figure 1: Simplified overview of the in-core part of an Intel Ice Lake server micro-architecture based on vendor data [11].

further divide the execution in two parts: (a) The *front end*, including instruction fetching and decoding as well as micro- and macro-operation fusion, and (b) the *back end* taking care of register allocation, instruction scheduling, the actual execution of instructions, and their retirement. It is common practice to model the in-core execution of a modern out-of-order (OoO) CPU with a *port model*. This means that each functional unit (such as ALU, LOAD unit, STORE unit, ...) is assigned to a port on which one instruction per cycle can be dispatched. This is independent of the details of the insructions, such as the number of pipeline stages or its maximum throughput. A simplified sketch of the front end and back end of the Intel *Golden Cove* micro-architecture as used by the *Ice Lake* processor can be found in Figure 1.

For an in-core performance model of a loop kernel, we make the following assumptions:

- We have a port model for the target micro-architecture
- There are no control flow branches inside of the loop
- The loop runs in a steady state, i.e., there are no wind-up or wind-down phases
- The CPU is capable of perfect OoO scheduling, i.e., there is no imbalance introduced by reservation stations
- All data resides in the L1 cache, i.e., there is no data delay from other memory hierarchy levels

1.2 In-core performance modeling approaches

There are several possible approaches for modeling the in-core performance of a loop kernel. The simplest way would be to use the peak floating-point performance as an upper limit as done by the Roofline modelexplained in Section 1.1. Given the fact that not every code solely consists of fused multiply-add (FMA) instructions, this ceiling is in almost all cases far away from the actual performance. Another upper-limit approach is to estimate the throughput of all instructions executed and take the execution time (reciprocal throughput) of the port with the highest pressure, i.e., the port port taking the longest time to finish all assigned instructions of the kernel, as a lower limit for the execution time of the entire kernel. While being more accurate than the previous approach, no dependencies between any instructions are considered. This assumption often proves to be valid since in practice multiple iterations of a loop kernel can overlap and dependencies within a loop (*intra-loop dependencies*) become irrelevant. This changes, however, if there are dependencies between different loop iterations. These *loop-carried dependencies* cannot be overlapped and thus represent are more realistic lower bound of the execution time. Finally, in contrast to the solutions looked at so far, the longest chain (time-wise) of dependent instructions can form a pessimistic, i.e., upper bound of the in-core execution time. If none of the individual loop iterations can overlap, this chain of instructions (the *critical path*) always has to be processed sequentially and, therefore, can give one additional insight on how many loop iterations must be kept in-flight for the processor to overlap delays created by intra-loop dependencies.

1.3 Open-Source Architecture Code Analyzer (OSACA)

The tool used in this tutorial for an in-core runtime prediction of assembly code is the Open-Source Architecture Code Analyzer (OSACA) [13, 14]. It is open-source[1] and supports various Intel (Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake-X, Cascade Lake, Ice Lake client, Ice Lake server), AMD (Zen 1, Zen 2, Zen 3), and ARM-based (Marvell Thunder X2, ARM Cortex A72/AWS Graviton, ARM Neoverse N1/AWS Graviton2, HiSilicon TaiShan v110, Fujitsu A64FX) micro-architectures. It is inspired by Intel's proprietary code analyzer IACA [9], which reached its end of life in 2019. OSACA parses x86 AT&T or AArch64 assembly code and applies the port model of the target micro-architecture to the instructions to calculate the throughput for each port and detect loop-carried dependencies and the critical path of the kernel. With this information, the user can get insight on possible bottlenecks and upper and lower bound for execution times:

$$T_{\text{lower}} = \max\left(\max(T_{\text{TP}_{\text{rcp}}\_\text{p0}}, T_{\text{TP}_{\text{rcp}}\_\text{p1}}, ..., T_{\text{TP}_{\text{rcp}}\_\text{pN}}), T_{\text{LCD}}\right)$$

$$T_{\text{upper}} = T_{\text{CP}}$$

Here, T_{lower} and T_{upper} are the estimated lower- and upper-bound runtime of a loop kernel, respectively, $T_{\text{TP}_{\text{rcp}}\_\text{p}i}$ is the reciprocal throughput prediction of port i, T_{LCD} the estimated execution time of the longest loop-carried dependency, and T_{CP} the estimated execution time of the critical path. OSACA's database is fed with with empirical data from micro-benchmark tools such as asmbench [8] and ibench [10], as well as data from uops.info [1]. At the time of the writing, OSACA does not support any modeling of the front end. Besides IACA, which only supports Intel micro-architectures, there are various other tools that are not used in this tutorial but support a wide range of micro-architectures (LLVM-MCA [4]), provide a

[1] https://github.com/RRZE-HPC/OSACA

```
1    double *a, *b, *c, s;
2
3    for (int i=0; i<N; ++i) {
4        a[i] = b[i] + s * c[i];
5    }
```

Listing 1: C loop body of STREAM TRIAD

highly accurate model of both front and back end (uiCA [2], currently only for Intel architectures), additionally produce an analysis about the code quality and vectorization (CQA [5]), or are based on machine learning (Ithemal [16]). None of them is, however, integrated with the Compiler Explorer (see below).

In the following we show a simple example of the usage of OSACA by analyzing the TRIAD kernel of the STREAM benchmark [15], shown in Listing 1. Assembly code for the loop body with AVX-512 is shown in Listing 2. After loading eight elements of the b array into zmm1 in line 2, eight elements of the c array are loaded in line 3, multiplied with the (register-wide broadcast) scalar value in zmm2, and added to the previously loaded vector. The result is stored back into in the a array in line 4. The three remaining lines are loop control instructions. It is important to notice that one assembly loop iteration amounts to eight C-language loop iterations due to vectorization.

The OSACA analysis of the assembly code can be found in Listing 3. The first ten columns show the reciprocal throughput of ports 0 to 9, respectively. The lower bound throughput estimate would therefore be 1 cy due to ports 2 and 3. Column "CP" reports the critical path and column "LCD" the longest loop-carried dependency, also being 1 cy and therefore not changing the lower-bound estimate. Thus, OSACA's analysis would result in 1 cy per eight (high-level) iterations.

1.4 Compiler Explorer

The Compiler Explorer [7] is an open-source interactive compiler exploration website[2] which makes it easy to see how code looks as compiled with different compilers and flags, compare the generated assembly, and even allows execution of the code. Since 2020, OSACA is integrated as an analysis tool in Compiler Explorer, making it straightforward to apply an in-core analysis in the web browser without any tool installed locally. For the tutorial, we deploy a local version of the Compiler Explorer running in a container on our university cluster. This allows us to fix the clock frequency of the CPUs and access the hardware counters for accurate measurements and, therefore, minimize the requirement of the tutorial attendees to a modern web browser and a working internet connection[3].

[2]Available at https://godbolt.org/

```
1  ..B2.42:
2    vmovups     (%r14,%rdx,8), %zmm1
3    vfmadd213pd (%r15,%rdx,8), %zmm2, %zmm1
4    vmovupd     %zmm1, (%r12,%rdx,8)
5    addq        $8, %rdx
6    cmpq        %rsi, %rdx
7    jb          ..B2.42
```

Listing 2: AVX-512 assembly loop body of STREAM TRIAD

2 TUTORIAL OUTLINE

In the following, we breifly describe the outline and the most important parts of the tutorial in the order it is taught to the attendees.

2.1 Basic processor and core architecture

We introduce the principal concepts of modern OoO processors based on the Intel Ice Lake server micro-architecture (Figure 1). The simplified workflow of an instruction from decoding to retiring is explained. We focus on the back end and introduce a general port model for OoO CPUs.

2.2 Terminology and code execution on out-of-order CPUs

We introduce important terms for the in-core performance analysis of loop kernels.

Throughput: The throughput describes the amount of work that can be done in a certain time, usually a cycle. We normally use the term *reciprocal throughput* to quantify the minimum number of cycles per instruction in steady-state, assuming the pipeline operates at capacity.

Latency: The latency of an individual instruction is defined by the time needed for the execution of it, i.e., the time between dispatch and being ready for retirement.

Critical Path: The critical path describes the time between the dispatch of the first instruction and the readiness to retire of the last instruction within the longest (time-wise) dependency chain within one kernel iteration, i.e., the sum of the latencies of the instruction in the longest dependency chain.

Loop-carried Dependency: Loop-carried dependencies are dependency chains that limit the overlap between different loop iterations. We usually refer to *the* loop-carried dependency as the longest of all existing dependency chains of a loop. This might be but is not necessarily the critical path or part of it.

Iteration: We highlight the difference between a high-level iteration of a loop kernel and an assembly loop iteration, which can consist of multiple high-level iterations due to unrolling and SIMD vectorization.

2.3 x86 ISA introduction

We give an introduction to the x86 instruction set architecture (ISA) and cover the differences between AT&T and Intel assembly syntax and their peculiarities, memory addressing modes, modern vector (SIMD) extensions (AVX), and masking.

2.4 Performance analysis of simple kernels

We deepen the knowledge of in-core performance analysis with x86 assembly by exercising it on several code examples. This includes the STREAM TRIAD (Listing 1), the DOT PRODUCT (Listing 4), and the computation of PI by integration (Listing 5).

[3]A simplified example without cycle count of the STREAM TRIAD is available at: https://godbolt.org/z/aW6qfno87

```
                    Port pressure in cycles
   |  0   |  1   |  2   |  3   |  4   |  5   |  6   |  7   |  8   |  9   ||  CP  |LCD|
---------------------------------------------------------------------------------------
 2 |      |      |      |      |      |      |      |      |      |      ||      |   |   ..B2.42:
 3 |      |      | 0.50| 0.50|      |      |      |      |      |      || 5.0  |   |   vmovups      (%r14,%rdx,8), %zmm1
 4 | 0.50|      | 0.50| 0.50|      | 0.50|      |      |      |      || 4.0  |   |   vfmadd213pd (%r15,%rdx,8), %zmm2, %zmm1
 5 |      |      |      |      | 0.50|      |      | 0.50| 0.50| 0.50|| 0.0  |   |   vmovupd      %zmm1, (%r12,%rdx,8)
 6 | 0.25| 0.25|      |      |      | 0.25| 0.25|      |      |      ||      | 1 |   addq         $8, %rdx
 7 | 0.00| 0.50|      |      |      | 0.00| 0.50|      |      |      ||      |   |   cmpq         %rsi, %rdx
 8 |      |      |      |      |      |      |      |      |      |      ||      |   | * jb           ..B2.42

     0.75   0.75   1.00   1.00   0.50   0.75   0.75   0.50   0.50   0.50    9      1
```

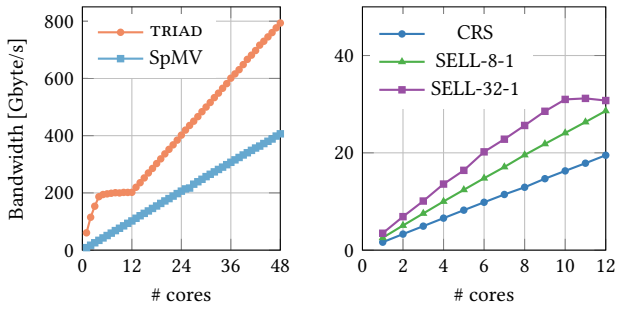
Listing 3: STREAM TRIAD analyzed by OSACA with `-arch=icx`.

```
1    double *a, *b, s;
2
3    for (int i=0; i<N; ++i) {
4        s = s + a[i] * b[i];
5    }
```

Listing 4: C loop body of the DOT PRODUCT kernel

```
1    double delta_x = 1./n;
2    double sum     = 0.0;
3
4    for (int i=0; i<n; i++) {
5        x    = (i + 0.5) * delta_x;
6        sum += (4.0 / (1.0 + x * x));
7    }
```

Listing 5: C loop body of the PI kernel



(a) Scaling of STREAM TRIAD and CRS-SpMV with the HPCG matrix using GCC 10.1.1. Working set size: 4 GB for TRIAD, 128^3 for HPCG.

(b) Scaling performance on one Core-Memory Group (CMG) for SpMV in SELL-C-σ and CRS format with the HPCG matrix, compiled with FCC 4.4.0a.

Figure 2: Scaling runs on Fujitsu FX700. Data taken from [3].

2.5 OSACA introduction

We introduce the OSACA tool and show how to use it both as standalone tool and integrated in the Compiler Explorer. We explain the usage of the Compiler Explorer and how we can use OSACA to pinpoint in-core bottlenecks of loop kernels. The attendees can validate their analysis by using a server configured for cycle-accurate measurements.

2.6 In-core analysis for Arm CPUs

We introduce the Fujitsu A64FX core architecture together with the basic principles of the AArch64 ISA. The attendees will understand the differences to the x86 ISA, enhanced memory addressing, vectorized ARM assembly (NEON and SVE), and masking.

2.7 Case studies and hands-on

After doing performance analysis on simple kernels, we show the importance and potential of performance engineering on real-life code with several more complex case studies:

- Sparse Matrix-Vector (SpMV) Multiplication on Fujitsu A64FX
- Lattice Quantum Chromodynamics (QCD) on Fujitsu A64FX
- 2D Gauss-Seidel on Intel Ice Lake

We cover the SpMV on A64FX as example of the three case studies in detail in the next section.

3 SPARSE MATRIX-VECTOR MULTIPLICATION (SPMV) ON A64FX

Figure 2a shows a scaling run for the STREAM TRIAD (Listing 1) benchmark and a SpMV ($b[:] = A[:,:] * x[i[:]]$) using the HPCG matrix in compressed row storage (CSR) format.

Compared to the STREAM TRIAD, which can saturate the memory bandwidth already with four cores, SpMV shows a factor of 3 lower single-core bandwidth and can thus not achieve bandwidth saturation. An in-depth analysis identifies the short inner loops during SpMV as potential bottleneck; a runtime estimate with OSACA reveals that the high latency of the fused multiply-add instruction (`fmla`, 9 cy) and the horizontal 512-bit ADD (`faddv`, 49 cy) are the culplrits. Changing the matrix format from CSR to the SIMD-friendly SELL-C-σ [12] makes the horizontal ADD obsolete and allows efficient vectorization. With sufficient unrolling, this results in a reduction of the `fmla` latency impact, which is shown in Figure 2b. With $C = 32$, the FX700 can saturate the bandwidth at ten cores.

4 CONCLUSIONS

We provide a tutorial about in-depth core-level performance engineering on different hardware platforms. After the exercises, the attendees will be experienced enough to carry out in-core performance engineering themselves and have knowledge about x86 and ARM assembly and how to pinpoint bottlenecks in assembly code.

REFERENCES

[1] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[2] Andreas Abel and Jan Reineke. 2022. uiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, USA, June 27-30, 2022 (ICS '22)*, Lawrence Rauchwerger, Kirk Cameron, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos (Eds.). ACM, 1–12. https://dl.acm.org/doi/pdf/10.1145/3524059.3532396

[3] Christie Alappat, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, Nils Meyer, and Tilo Wettig. 2020. Performance Modeling of Streaming Kernels and Sparse Matrix-Vector Multiplication on A64FX. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 1–7. https://doi.org/10.1109/PMBS51919.2020.00006

[4] Andrea Di Biagio. 2023. llvm-mca – LLVM Machine Code Analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html

[5] Andres S. Charif-Rubial, Emmanuel Oseret, José Noudohouenou, William Jalby, and Ghislain Lartigue. 2014. CQA: A code quality analyzer tool at binary level. In *2014 21st International Conference on High Performance Computing (HiPC)*. 1–10. https://doi.org/10.1109/HiPC.2014.7116904

[6] Lieven Eeckhout. 2010. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–145. https://doi.org/10.2200/S00273ED1V01Y201006CAC010

[7] Matt Godbolt. 2012. Compiler Explorer. https://godbolt.org/

[8] Julian Hammer, Georg Hager, and Gerhard Wellein. 2018. OoO Instruction Benchmarking Framework on the Back of Dragons. (2018). https://sc18.supercomputing.org/proceedings/src_poster/src_poster_pages/spost115.html SC18 ACM SRC Poster.

[9] Israel Hirsh and Gideon S. 2012. Intel® Architecture Code Analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer

[10] Johannes Hofmann. 2017. ibench - Instruction Benchmarks. https://github.com/RRZE-HPC/ibench

[11] Intel Corporation 2023. *Intel® 64 and IA-32 Architecture Optimization Reference Manual*. Intel Corporation. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual

[12] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423. https://doi.org/10.1137/130930352 arXiv:https://doi.org/10.1137/130930352

[13] J. Laukemann, J. Hammer, G. Hager, and G. Wellein. 2019. Automatic Throughput and Critical Path Analysis of x86 and ARM Assembly Kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 1–6. https://doi.org/10.1109/PMBS49563.2019.00006

[14] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. 2018. Automated Instruction Stream Throughput Prediction for Intel and AMD Microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 121–131. https://doi.org/10.1109/PMBS.2018.8641578

[15] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25. http://cs.virginia.edu/stream

[16] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*. PMLR, 4505–4515.

[17] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76. https://doi.org/10.1145/1498765.1498785