OoO Instruction Benchmarking Framework on the Back of Dragons

Poster Summary

Julian Hammer
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
julian.hammer@fau.de

Gerhard Wellein (advisor)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
gerhard.wellein@fau.de

Georg Hager (advisor)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
georg.hager@fau.de

ABSTRACT

In order to construct an accurate instruction execution model for modern out-of-order micro architectures, an accurate description of instruction latency, throughput and concurrency is indispensable. Already existing resources and vendor provided information is neither complete nor detailed enough and sometimes incorrect. We therefore proclaim to deduct this information through runtime instruction benchmarking and present a framework to support such investigations based on LLVM's just-in-time and cross-platform compilation capabilities.

`asmbench` abstracts instructions, operands and dependency chains, to easily construct the necessary benchmarks. The synthesized code is interactively compiled and executed using the `llvmlite` library, which is based on LLVM's C-API. `asmbench` offers a command line, as well as a programming interface.

Unlike other approaches, we do not rely on model specific performance counters and focus on interoperability and automation to support quick modeling of future microarchitectures.

CCS CONCEPTS

• **General and reference → Performance**; *Measurement*; *Experimentation*; Metrics; • **Computer systems organization → Architectures**; **Pipeline computing**; • **Software and its engineering → Software performance**; *Assembly languages*; *Compilers*; Just-in-time compilers;

KEYWORDS

benchmarking, performance modelling, architecture modelling, assembly

1 MOTIVATION

Performance models are vital during development of performance critical applications and compilers to evaluate optimization strategies. Accurate models require a deep understanding of execution and data behavior of micro architectures. When focusing on execution, modeling may be as simple as counting floating point operations and dividing them by a processor's peak performance [9], or as complex as cycle accurate full-system simulation [7].

In our research on performance modeling, we are building the automated performance modeling toolkit `kerncraft` [8], which allows performance-aware developers to quickly generate kernel-specific performance models. We are currently basing the execution predictions on Intel's Architecture Core Analyzer (IACA) [3], which is based on an undisclosed model and only supports Intel architectures.

To lift `kerncraft`'s current limitation to Intel architectures–due to the limitations of IACA–an open-source IACA-alternative is needed. Such a tool will require detailed information about the available instructions and scheduling model, to accurately predict execution times, dependencies and concurrency.

Gathering a comprehensive collection of instruction performance data on micro architectures is thus the first stepping stone to building an open and disclosed execution performance model. We present a framework to support such endeavors: `asmbench`.

2 RELATED WORK

Instruction performance information is already available, but the two well-known sources are insufficient: Intel publishes performance information in the Architectures Optimization Reference Manual [1], it is neither machine readable nor complete, and sometimes even incorrect. Agner Fog compiles very useful tables, based on his own observation of architectures, but these "Instruction Tables" [5] are also incomplete and not easily reusable for automated predictions, due to the format and ambiguities.

Others have made attempts to automated instruction benchmarking, the three most notable examples are: `llvm-exegesis` [6], a recent addition to LLVM, which validates instruction performance data available to the compiler. It is based on model specific performance counters, which make it hard to port to new architectures. `mubench!`[2], an abandoned open-source project from 2008, which focused on x86, is also not portable. `ibench` [4] and `likwid-bench` [10] provide the framework to run benchmarks but aim toward manual benchmarking and x86 rather then automatically evaluating a complete micro architecture.

3 APPROACH

To get a complete model of an architecture, a high degree of automation is necessary. Therefore, benchmarks need to be generated automatically from already existing instruction set information and all boilerplate code used for instrumentation needs to be architecture independent.

We achieve this through LLVM's intermediate representation and backend instruction database "TableGen". For each instruction, where input and output operands can be the same register type, a throughput and latency benchmark are generated. In case of latency, a long chain of instructions is created, where each output is

directly consumed as input for the next instruction. For throughput, multiple such chains are created, each independent of the others.

By wrapping the benchmarks in a sufficiently long loop, until a minimal runtime of 0.2 s is surpassed, accurate measurements can be obtained. For precise results, it is also necessary to deactivate any frequency scaling or "turbo mode" on the processor. This is sometimes tricky, e.g., our AMD Zen system would always go into turbo unless the clock was set to a frequency bellow its nominal base clock.

Since each instruction uses some number of execution units, other instructions may run independently in parallel or they may share the same resources, leading to resource conflicts. To quantify resource conflicts, all possible instruction pairs are benchmarked. By comparing the reciprocal throughput (runtime per instructions executed) of combined instructions ($\mathrm{TP}^{-1}(A+B)$) to individual execution ($\mathrm{TP}^{-1}(A)$ and $\mathrm{TP}^{-1}(B)$), we are able to deduce resource conflicts:

$$\frac{\mathrm{TP}^{-1}(A+B) - \max(\mathrm{TP}^{-1}(A), \mathrm{TP}^{-1}(B))}{\min(\mathrm{TP}^{-1}(A), \mathrm{TP}^{-1}(B))} = \begin{cases} \gg 1 & \text{additional} \\ & \text{overhead} \\ \approx 1 & \text{no overlap} \\ & \text{conflict} \\ \approx 0 & \text{complete overlap} \\ \ll 0 & \text{elimination} \end{cases}$$

The resource conflict metric tells us if two instructions influence each others runtime negatively ($rc \gg 0$). A resource conflict suggests that both instructions shared an execution unit. This metric is to be interpreted as qualitative information, since errors from individual measurements accumulate.

4 SUMMARY AND FUTURE WORK

To illustrate the possibilities, we performed our analysis methodology on 24 x86 instructions. For each instruction we present throughput and latency with comparison to Agner Fog's "Instruction Tables" [5] and Intel's Documentation – where available. We also present our resource conflict metric for all pair-wise combinations of the 24 instructions. From the presented date we can already see that other sources are incomplete and sometimes incorrect. It is also possible to see a pattern emerging from resource conflicts, where groups of instructions form, which have no overlap, these instructions are implemented using the same hardware resources and therefore serialize on some level in the processor.

All measurements were performed on dedicated hardware, with fixed frequency and disabled turbo mode. Out of multiple runs the lowest runtime was used, since maximum throughput and latency is what we seek.

This is all ground work for our modeling ambitions to build a complete in-core model from empirical data and port this model to many architectures.

In future, the scope of this framework needs to be broadened further to encompass all relevant effects, instructions and architectures (e.g., combined load and arithmetic instructions are currently not supported).

ACKNOWLEDGMENTS

The work is supported by the German Ministry of Education and Research METACCA project.

REFERENCES

[1] [n. d.]. Intel 64 and IA-32 Architectures Optimization Reference Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
[2] 2006. mubench – in-depth benchmark for x86 and x84-64 processors. http://mubench.sourceforge.net/
[3] 2017. Intel Architecture Code Analyzer. https://software.intel.com/en-us/articles/intel-architecture-code-analyzer
[4] 2018. ibench – Measure Instruction Latency and Throughput. https://github.com/hofm/ibench
[5] 2018. Instruction Tables. http://www.agner.org/optimize/instruction_tables.pdf
[6] 2018. llvm-exegesis – LLVM Machine Instruction Benchmark. https://llvm.org/docs/CommandGuide/llvm-exegesis.html
[7] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, David A. Wood, Bradford Beckmann, Gabriel Black, and et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (8 2011), 1. https://doi.org/10.1145/2024716.2024718
[8] Julian Hammer, Georg Hager, Jan Eitzinger, and Gerhard Wellein. 2015. Automatic loop kernel analysis and performance modeling with Kerncraft. *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems - PMBS '15* (2015). https://doi.org/10.1145/2832087.2832092
[9] H. T. Kung. 1986. Memory Requirements for Balanced Computer Architectures. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA '86)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 49–54. http://dl.acm.org/citation.cfm?id=17407.17362 DOI: 10.1145/17356.17362.
[10] Jan Treibig, Georg Hager, and Gerhard Wellein. 2012. likwid-bench: An Extensible Microbenchmarking Platform for x86 Multicore Compute Nodes. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Mueller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–36.