Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks

Charith Mendis, Alex Renda, Saman Amarasinghe, Michael Carbin
MIT CSAIL
charithm@mit.edu, {renda, saman, mcarbin}@csail.mit.edu

ABSTRACT

Predicting the number of clock cycles a processor takes to execute a block of assembly instructions in steady state (the *throughput*) is important for both compiler designers and performance engineers. Building an analytical model to do so is especially complicated in modern x86-64 Complex Instruction Set Computer machines with sophisticated processor microarchitectures in that it is tedious, error-prone, and must be manually updated. In this paper we present Ithemal, the first tool which *learns* to predict the throughput of a set of instructions. Ithemal uses a hierarchical LSTM–based approach to predict throughput of instructions in a basic block. We show that Ithemal is more accurate than state-of-the-art analytical tools (less than half the error) currently used in compiler backends and static machine code analyzers. Ithemal is also able to predict throughput just as fast as the analytical tools, and is easily ported across a variety of microarchitectures with minimal effort.

1. INTRODUCTION

The *throughput* of a sequence of instructions—the number processor clock cycles taken to execute the sequence when looped in steady state—determines how fast those instructions can process data. Accurately predicting the throughput of a basic block is important for many systems to be able to predict and optimize runtime performance, for example in compiler algorithms such as genetic algorithm based register allocation [1] and reinforcement learning based instruction scheduling [2].

The alternative – measuring throughput on demand by executing the basic block – is too expensive for most compilers. In practice, most systems employ analytical models to predict throughput. For instance, the LLVM compiler team [3] recently merged[1] a command-line tool, llvm-mca [4], that exposes a machine model for throughput estimation. Intel has also released a closed-source machine code analyzer, IACA [5], which relies on internal knowledge of Intel's processor design. These models are typically an order of magnitude faster than measuring a basic block's throughput. However, manually writing an accurate and complete model is tedious, error-prone, and exceedingly difficult without knowledge of the exact mechanisms of the processor.

In the hunt for *accuracy*, developers build complicated models which must make significant tradeoffs with the model's *portability* and *speed*.

[1]lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html

- **Accuracy.** Modern x86-64 Complex Instruction Set Computer (CISC) processors contain many hardware optimizations that significantly complicate building accurate analytical models. They execute sequences of instructions in heavily piplined, out-of-order and superscalar execution units with latent vendor-specific optimizations. This makes the prediction problem highly complex and non-linear.
- **Portability.** Manually writing a throughput estimator to support different microarchitectures requires rewriting instruction tables, resource utilization charts, and modeling microarchitectural optimizations, all of which are tedious and error-prone. Ideally, the throughput estimator should be able to automatically capture such intricacies with minimal human intervention.
- **Speed.** A throughput estimator also needs to be fast. Compilers need to search through many code blocks before emitting the fastest version of a given instruction sequence. Running the basic blocks to get the ground truth throughput requires sandboxing and many iterations of execution to arrive at a consistent steady-state throughput estimate, which can be impractical for systems performing fast searches [6].

1.1 Ithemal: A Data Driven Approach

In this paper we introduce *Ithemal (Instruction THroughput Estimator using MAchine Learning)*, which takes a novel data–driven approach to predicting throughput for a block of instructions, inspired by recent advances in Deep Neural Networks (DNNs). Ithemal models the throughput estimation problem as a regression task and leverages a DNN to learn to predict throughput by using a large corpus of labeled data, mapping assembly sequences to real valued throughputs. More concretely, Ithemal uses a hierarchical multiscale RNN [7, 8, 9], which generates an independent embedding for each instruction, then sequentially combines the instruction embeddings to predict throughput.

We show that Ithemal's learned model is significantly more accurate than the analytical models, dropping the mean absolute percent error by more than 50% across all benchmarks, while still delivering fast estimation speeds.

To generate high-quality predictions, Ithemal needs only training data and a specification of the ISA, including the specification of instructions and their explicit and implicit operands (for instance, the instruction `push rax` in x86-64 pushes the register `rax` on to the stack and also *implicitly*

An extended manuscript is at arxiv.org/abs/1808.07412

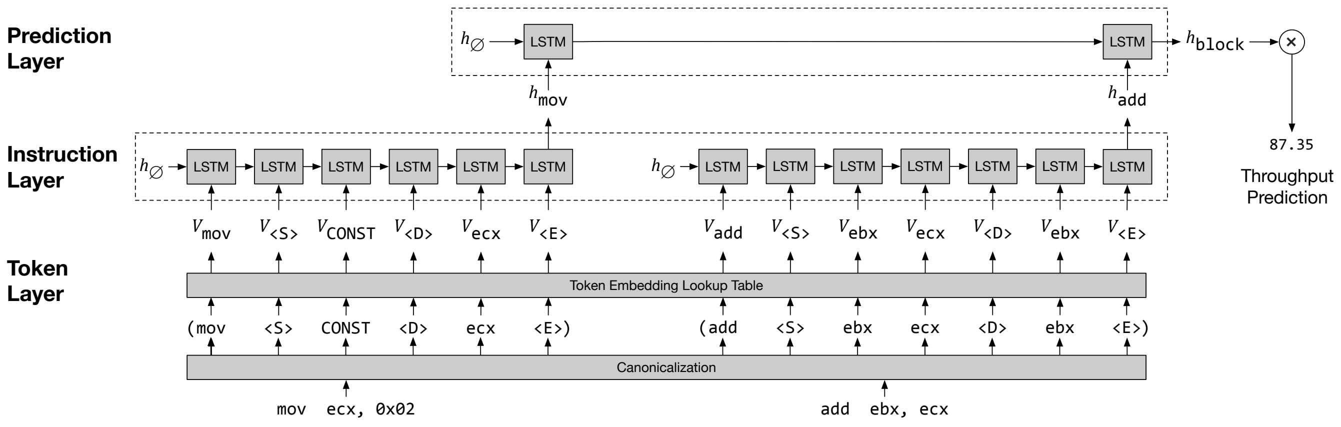
Prediction Layer
Instruction Layer
Token Layer
LSTM
$h_{\varnothing}$
h_{block}
h_{mov}
h_{add}
87.35
Throughput Prediction
v_{mov} $v_{\text{<S>}}$ v_{CONST} $v_{\text{<D>}}$ v_{ecx} $v_{\text{<E>}}$
v_{add} $v_{\text{<S>}}$ v_{ebx} v_{ecx} $v_{\text{<D>}}$ v_{ebx} $v_{\text{<E>}}$
Token Embedding Lookup Table
(mov <S> CONST <D> ecx <E>)
(add <S> ebx ecx <D> ebx <E>)
Canonicalization
mov ecx, 0x02
add ebx, ecx

Figure 1: Ithemal System Architecture

modifies the stack pointer register, `rsp`). Unlike analytical models, Ithemal learns any salient microarchitectural details that contribute to throughput on its own, without any explicit specification or modeling.

For example, consider the instruction `vxorps xmm0, xmm0, xmm0` which zeros out register `xmm0`. Intel microprocessors identify such zeroing idioms and execute it in a separate optimized data path. The true throughput for 100 iterations of this instruction is measured at 32 clock cycles. However, llvm-mca reports a throughput of 100, where as Ithemal which is intrinsically learned from data predicts 35 cycles. This simple example is one of many where the data-driven model is able to saliently capture microarchitectural optimizations.

Ithemal demonstrates that future systems can leverage data–driven techniques to either augment or fully replace manually developed throughput estimators. We have open-sourced our implementation of Ithemal at `https://github.com/psg-mit/Ithemal` in the hope that performance engineers and compiler designers can use and improve upon our approach.

2. MODEL ARCHITECTURE

Figure 1 presents the high-level design of Ithemal's approach. We model the problem of throughput estimation as a regression problem: given the assembly input, Ithemal predicts the throughput of the instruction sequence as a real-valued number. At the core of Ithemal is a hierarchical multi-scale RNN [10, 11] that sequentially processes all instructions in the basic block and outputs an embedding, which Ithemal then uses to directly estimate the throughput. Altogether, we decompose the end-to-end model into the following stages: *canonicalization*, *embedding* and *estimation*.

2.1 Canonicalization

The canonicalization stage converts the assembly input into a more structured form, dictated by the syntax of the assembly instructions. Ithemal takes a compiled assembly block, disassembles it, and maps it to a list of instructions. Each instruction consists of a list of tokens representing its operation code (opcode, e.g. `add`), source operands, and destination operands, separated by distinguished delimiter tokens. For example, consider the instruction `mul ecx`, which multiplies the value in register `ecx` with `eax`, and places the result into registers `edx` and `eax`. Note that the source operand `eax` and both of the destination operands `eax` and `edx` are implicit in the Intel syntax `mul ecx`. The final canonicalized set of tokens for the instruction is:

$$(\texttt{mul}, <S>, \texttt{eax}, \texttt{ecx}, <D>, \texttt{edx}, \texttt{eax}, <E>)$$

where the bracketed tokens are the delimiters representing the break between the opcode, source, and destination operands.

Assembly code permits constants and memory operands. We map all constants (e.g. integer constants, absolute memory addresses, etc.) to a single `CONST` token. We demarcate memory operands (consisting of a base address, and an optional offset and displacement) by surrounding them with `<M>` and `</M>` delimiter tokens.

2.2 Embedding

Ithemal's embedding stage takes a canonicalized token stream of instructions, and for each instruction produces an *embedding*: a representation of an instruction as a real-valued vector in a high-dimensional space. The first step is the *token layer*, which maps a given token to an embedding. We implement the token layer by mapping each token in the sequence to an n-dimensional vector by learning a linear transformation of the one-hot token vectors (this is equivalent to learning a lookup table).

Ithemal then maps the sequence of token embeddings to an embedding for each instruction in the basic block. We call this the *instruction layer*. Because each instruction can have a variable number of tokens depending on its number of source and destination operands, the size of the input to the embedding stage is variable. We therefore implement the instruction layer with a sequential Recurrent Neural Network (RNN) architecture with Long Short Term Memory (LSTM) [12] cells.

Figure 1 presents the operation of our RNN-based instruction embedding approach on a small example. The bottom-most row shows the assembly input. The second row shows the tokens for each instruction. The third row (the token layer) shows the *token embeddings*, e.g. $v_{\texttt{mov}}$, which are mapped directly from each syntactic token. The fourth row (the instruction layer) shows applying a LSTM to reduce the token embeddings into the final instruction embedding, $h_{\texttt{mov}}$.

2.3 Prediction

The final prediction comes from the *prediction layer*, which maps a basic block (a sequence of instruction embeddings) to a throughput value. This is again implemented with an RNN with LSTM cells, which has entirely disjoint weights from the LSTM in the instruction layer. This corresponds to the topmost layer in Figure 1. Using the final output from the instruction LSTM ($h_{\texttt{block}}$), Ithemal predicts the basic block's throughput with a linear layer. Specifically, Ithemal computes $w \cdot h_{\texttt{block}} + b$, where w is a learned weight vector and b is a bias. This produces a final real-valued number that represents the network's throughput prediction.

3. DATA AND TRAINING

We collected a dataset of basic blocks from well-known programs and benchmark suites, and timed them with a procedure that matches the assumptions of the baseline analytical models. We then train Ithemal using standard supervised learning techniques.

3.1 Dataset

We designed the dataset to include a diverse set of applications with different performance characteristics while covering a wide range of x86-64 instructions. It consists of performance critical applications used for benchmarking compiler optimizations (e.g., SPEC2006 [13], SPEC2017 [14], NAS [15] benchmarks) as well as end user applications used in day-to-day computing (e.g., firefox, open-office, gimp)

To extract each application's basic blocks, we first compile each application using GCC 4.9.4 with the `-O3` optimization level targeting an Intel Haswell processor. Next, we run each application under its standard inputs and dump the encoded bytes of the executed x86-64 basic blocks using DynamoRIO [16]. Finally, we de-duplicate the dataset by removing basic blocks with same encoded byte patterns.

3.2 Throughput Profiling

IACA and llvm-mca predict the steady-state throughput of a basic block, under the assumptions that all memory accesses result in L1 cache hits and that the execution environment is non-preemptive. To collect compatible throughput numbers, we profile the execution of a loop that executes each basic block in isolation 100 times (enough to reach the steady-state behavior; 100 iterations is also the default value used by llvm-mca). We measure throughput in terms of clock cycles. Our timing script ensures that almost all memory accesses have a L1 cache hit. Additionally, we measure L1 instruction and data cache misses and software context switches to detect and filter out invalid executions that do not conform to the assumptions made by IACA and llvm-mca in their predictions.

Using this methodology, we collected valid throughput values for the Intel Ivy Bridge (Intel(R) Xeon(R) CPU E5-2695 v2), Haswell (Intel(R) Xeon(R) CPU E5-2680 v3) and Skylake (Intel(R) Xeon(R) W-2123 CPU) microarchitectures. Data collection takes approximately 4-5 days for each microarchitecture. The final Haswell dataset, which is de-duplicated across benchmarks, constitutes 1,416,473 unique basic blocks.

3.3 Training and Methodology

We implemented our neural network model in PyTorch (0.4.0a0+59bda9a). The learnable parameters in Ithemal include the token embeddings, the token LSTM and instruction LSTM parameters, and the affine coefficients in the final linear layer. For our loss function we use a normalized error metric, based on the L1 norm:

$$\mathscr{L}(\texttt{pred}, \texttt{actual}) = \frac{|\texttt{pred} - \texttt{actual}|}{\texttt{actual}}$$

We randomly assign 80% of the collected blocks to the train set and 20% to the test set. We use Asynchronous Stochastic Gradient Descent [17, 18] with a batch size of 4 to train the model, using 6 parallel trainers, an initial learning rate of 0.1, and a momentum of 0.9. Each epoch after the first two, we decay the learning rate by a factor of 1.2. To handle `NaN` gradients, any trainer that encounters a `NaN` is halted for the remainder of the epoch. Training runs until all trainers have halted.

4. EVALUATION

We have evaluated Ithemal against two state-of-the-art, hand-written analytical models: IACA [5] (v3.0-28-g1ba2cbb) and llvm-mca [4] (LLVM 8.0.0). Both of these models are designed to model the complexities of modern processors (including pipelining, superscalar, and out-of-order units). We show that our data–driven model beats the accuracy of these sophisticated hand-written models (Section 4.1) while maintaining just as fast prediction speeds. Further, we show that our approach is portable across different microarchitectures in Section 4.2 by showing that Ithemal learns a model that outperforms IACA and llvm-mca without any neural network architecture or hyperparameter modifications.

4.1 Accuracy

We evaluate the accuracy of each model against the actual throughput values for Intel's Haswell, Ivy Bridge, and Skylake microarchitectures. The version of IACA we use does not support throughput estimation for Ivy Bridge; we therefore evaluate accuracy only for Ithemal and llvm-mca for Ivy Bridge. We prepared train and test sets for each microarchitecture according to the description in Section 3.3.

Table 1 presents the results of our accuracy comparison. We report the average error with respect to the ground truth of each tool for each microarchitecture. We also report both the Spearman and Pearson correlation of each tool's predictions with the ground truth.

Ithemal is more accurate in its throughput predictions for basic blocks across all three microarchitectures. Our model's predictions are closer to the ground truth than both IACA and LLVM in 74% of the blocks in the Haswell test set. Ithemal's predictions also have a higher correlation with the ground truth values for both the Spearman (rank correlation) and Pearson (linear correlation) metrics. The higher Spearman correlation is especially useful because it directly corresponds to higher utility for use within an optimizing compiler (such as an instruction scheduling pass). Specifically, compilers typically only need to determine which of two (or more) optimized configurations of a basic block is the fastest, and do not calculate each block's absolute performance.

Further, Ithemal's prediction speed (560 instructions/s) is as fast as llvm-mca (492 instructions/s) and IACA (541 instructions/s) in our measurements, and is significantly faster than empirical evaluation of basic blocks (13 instructions/s). Hence, Ithemal functions as an equivalently performant and more accurate drop-in replacement for llvm-mca and IACA in systems which only need throughput estimations, while still performing significantly faster than empirical evaluation.

Micro-architecture	Method	Error	Spearman Correlation	Pearson Correlation
Ivy Bridge	llvm-mca	0.181	0.902	0.777
	Ithemal	**0.089**	**0.955**	**0.913**
Haswell	llvm-mca	0.200	0.890	0.790
	IACA	0.209	0.917	0.833
	Ithemal	**0.089**	**0.960**	**0.918**
Skylake	llvm-mca	0.239	0.852	0.729
	IACA	0.167	0.926	0.835
	Ithemal	**0.079**	**0.960**	**0.895**

Table 1: Average error for different models and microarchitectures

4.2 Portability

We designed and trained Ithemal on Haswell and validated our architecture and hyperparameters by re-training on Skylake. Without any changes to its structure or training regime, we then trained and evaluated Ithemal on the Ivy Bridge dataset. Table 1 summarizes the average errors for each microarchitecture. Ithemal learns to estimate throughput values for each microarchitecture with a maximum average error of 0.089 across all datasets. The hand-written models exhibit a minimum average error of 0.167.

In sum, Ithemal provides state-of-the-art prediction performance; its results beat the baselines across the board. Moreover, Ithemal does so without requiring a user to provide information about the processor's underlying microarchitecture, whereas these analytical models require significant re-engineering for each microarchitecture of interest.

5. DISCUSSION

Compiler optimization passes rely on performance models of the target architecture to perform both back-end optimizations like instruction selection and register allocation, and middle-end optimizations like SLP vectorization [19], loop vectorization, loop unrolling, inlining, and more. Inaccurate performance models may drive the compiler into making sub-optimal decisions with little (or even negative) correlation with real-world timing. Unfortunately, as we have shown, analytical performance models of x86 code are highly inaccurate compared to learned solutions.

This is a serious problem for back-end optimizations, but it is exacerbated by the even simpler models often used in compiler middle-ends. For instance, LLVM's `TargetTransformInfo` interface defines a linear cost model which assumes that all instructions are independent and have no microarchitectural interactions. While Ithemal is not an IR-level tool, it does show that complex performance models of code are learnable from raw data without needing any baked-in domain knowledge.

Unfortunately, despite Ithemal outperforming other state-of-the-art analytical models, new performance models are not enough. Many cost-model-driven optimization algorithms rely on linear cost models, and do not have obvious analogues for nonlinear models like Ithemal or even llvm-mca or IACA. To fully take advantage of learned cost models, we must also develop new compilation techniques which can handle the relaxed assumptions inherent in more accurate cost models.

Further work is also necessary to relax even more of the assumptions in Ithemal. Ithemal, along with comparable analytical throughput prediction tools, only models the first-order performance characteristics of basic blocks. However, performance of memory bound applications depends heavily on memory access patterns. With emerging techniques that learn memory access patterns [20], we expect to be able to build more complete learned performance that model both compute and memory intensive workloads.

6. RELATED WORK

Apart from state-of-the-art tools like llvm-mca and IACA, other analytical models exist for throughput estimation [21] of instructions. OSACA [22] is an open source analytical model similar to llvm-mca and IACA, which automates some of the collection of the tabular data which is plugged into the model. There are also analytical models such as [23] to estimate throughput for multithreaded programs. Cycle–accurate simulators such as ZSim [24] and Marss [25] have a high start-up and runtime cost. All of these models require detailed processor modeling and considerable human effort.

There has been work on developing machine learning–based models for absolute and relative runtime estimation. [26] introduces sparse polynomial regression to predict execution time of programs by using a set of hand–crafted features of high level programs. [27] uses neural networks with hand-crafted features to estimate the speedup between two code sequences. GameTime [28, 29] uses SMT solvers to generate inputs and game theoretic approaches to predict the distribution of runtimes of programs. These models require manual feature engineering, and runtime predictions are done at a coarser granularity (e.g. at the full program level). In contrast, Ithemal automatically learns how to predict throughput of basic blocks with minimal architecture-specific knowledge embedded into the model.

Similar to basic block throughput estimation, various microarchitectural prediction tasks have been explored with machine learning. For example, sequential RNN models can be used for predicting memory access [20], and perceptron models can be used for branch prediction [30].

7. CONCLUSION

We present Ithemal, a data–driven system for basic block throughput estimation. Ithemal's accuracy surpasses that of state-of-the-art, hand-written analytical models; it achieves its accuracy by leveraging a deep neural network designed to capture the behavior of modern processors. Ithemal demonstrates that future compilation and performance engineering tools can be augmented with data–driven approaches to improve their performance and portability.

8. REFERENCES

[1] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, (New York, NY, USA), pp. 77–90, ACM, 2003.

[2] A. McGovern and E. Moss, "Scheduling straight-line code using reinforcement learning and rollouts," in *Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems II*, (Cambridge, MA, USA), pp. 903–909, MIT Press, 1999.

[3] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[4] A. Di Biagio and M. Davis, "llvm-mca," 2018.

[5] Intel, "Intel architecture code analyzer," 2017.

[6] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," *SIGPLAN Not.*, vol. 48, pp. 305–316, Mar. 2013.

[7] S. El Hihi and Y. Bengio, "Hierarchical recurrent neural networks for long-term dependencies," in *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, (Cambridge, MA, USA), pp. 493–499, MIT Press, 1995.

[8] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical multiscale recurrent neural networks," *ICLR'17*, 2017.

[9] L. Baraldi, C. Grana, and R. Cucchiara, "Hierarchical boundary-aware neural encoder for video captioning," *CVPR'17*, 2017.

[10] B. Shuai, Z. Zuo, G. Wang, and B. Wang, "Dag-recurrent neural networks for scene labeling," *CoRR*, vol. abs/1509.00552, 2015.

[11] X. Zhu, P. Sobhani, and H. Guo, "Dag-structured long short-term memory for semantic compositionality," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 917–926, 2016.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.

[13] SPEC, "Spec cpu2006 benchmark suite," 2006.

[14] SPEC, "Spec cpu2017 benchmark suite," 2017.

[15] A. S. D. NASA, "Nas c benchmark suite 3.0," 1991–2014.

[16] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, (New York, NY, USA), pp. 133–144, ACM, 2012.

[17] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[18] F. Niu, B. Recht, C. Re, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, (USA), pp. 693–701, Curran Associates Inc., 2011.

[19] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, (New York, NY, USA), pp. 145–156, ACM, 2000.

[20] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 1924–1933, 2018.

[21] T. M. Taha and D. S. Wills, "An instruction throughput model of superscalar processors," in *Rapid Systems Prototyping, 2003. Proceedings. 14th IEEE International Workshop on*, pp. 156–163, IEEE, 2003.

[22] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein, "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)*, pp. 121–131, Nov 2018.

[23] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model," in *HPCA-15 2009. IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 329–340, IEEE, 2009.

[24] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer architecture news*, vol. 41, pp. 475–486, ACM, 2013.

[25] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 1050–1055, IEEE, 2011.

[26] L. Huang, J. Jia, B. Yu, B. gon Chun, P. Maniatis, and M. Naik, "Predicting execution time of computer programs using sparse polynomial regression," in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 883–891, Curran Associates, Inc., 2010.

[27] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, (New York, NY, USA), pp. 131–142, ACM, 2007.

[28] S. A. Seshia and J. Kotker, "GameTime: A toolkit for timing analysis of software," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 388–392, March 2011.

[29] S. A. Seshia and A. Rakhlin, "Quantitative analysis of systems using game-theoretic learning," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. S2, p. 55, 2012.

[30] D. A. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 197–206, Jan 2001.