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

Charith Mendis \(^1\)  Alex Renda \(^1\)  Saman Amarasinghe \(^1\)  Michael Carbin \(^1\)

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 (CISC) machines with sophisticated processor microarchitectures in that it is tedious, error prone, and must be performed from scratch for each processor generation. 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 based on the opcodes and operands of instructions in a basic block. We show that Ithemal is more accurate than state-of-the-art hand-written tools currently used in compiler backends and static machine code analyzers. In particular, our model has less than half the error of state-of-the-art analytical models (LLVM's llvm-mca and Intel's IACA). Ithemal is also able to predict these throughput values just as fast as the aforementioned tools, and is easily ported across a variety of processor microarchitectures with minimal developer 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\(^1\) is an essential requirement in many systems, to be able to predict and optimize runtime performance. For instance, constraint–based register allocation and instruction scheduling (Lozano et al., 2012) relies on accurate throughput estimations, as do learning–based techniques like genetic algorithm based register allocation (Stephenson et al., 2003) and reinforcement learning based instruction scheduling (McGovern & Moss, 1999).

The alternative – measuring throughput on demand by executing the basic block – is too expensive for most compilers and learning–based solutions. In practice, most systems employ analytical models to predict throughput. For instance, the LLVM compiler team (Lattner & Adve, 2004) recently merged\(^2\) a command-line tool, llvm-mca (Di Biagio & Davis, 2018), that exposes a machine model for throughput estimation. Intel has also released a closed-source machine code analyzer, IACA (Intel, 2017), 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.

Accuracy. Modern x86-64 Complex Instruction Set Computer (CISC) processors contain many hardware optimizations that significantly complicate building accurate analytical models. In order to implement an instruction set architecture (ISA) like x86-64, processors actually implement an underlying microarchitecture, a physical implementation of the ISA specification. Processors translate instructions from the ISA to instructions in the latent microarchitectural language (termed micro-ops), then execute those micro-ops. The micro-ops may undergo optimizations such as micro-op fusion, in which micro-ops of different instructions may be combined together; out-of-order execution, in which instructions can be executed in any semantics–preserving order; register renaming, where false dependencies can be broken to enable more parallel execution; and many more vendor-specific optimizations. This makes the prediction problem highly complex and non-linear.

\(^1\)MIT CSAIL. Correspondence to: Charith Mendis <charithm@mit.edu>.
\(^2\)lists.llvm.org/pipermail/llvm-dev/2018-March/121490.html
**Portability.** While ISAs like x86-64 stay relatively stable, processor vendors release updated processor implementations with different microarchitectures every few years. For example, Intel released the Haswell and Skylake microarchitectures in 2013 and 2015 respectively for the x86-64 instruction set. Each microarchitecture of a processor family has its own quirks and intricacies. 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. This is complicated by the vast, incomplete, and incorrect documentation for many processors, where understanding of these behaviors has to be obtained by reverse-engineering the processor. 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 is impractical for real-time systems.

### 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 a hierarchical multiscale RNN (El Hihi & Bengio, 1995; Chung et al., 2017; Baraldi et al., 2017), 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 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.

In this paper, we present the following contributions:

- **Data–Driven Throughput Estimation.** We present Ithemal, the first system for data-driven basic block throughput estimation, using a hierarchical multiscale RNN.
- **Evaluation:** We demonstrate that Ithemal is more accurate than the state-of-the-art analytical throughput estimators, while still attaining fast estimation speeds. We also show that Ithemal’s design is portable across multiple processor microarchitectures.
- **DNN Architecture Exploration.** We demonstrate that the proposed hierarchical multiscale RNN outperforms many other choices, including other neural network architectures specifically tailored to mimic the dependency patterns of instructions within a basic block.
- **Open Source Implementation.** We have open-sourced our implementation of Ithemal at [https://github.com/psg-mit/Ithemal](https://github.com/psg-mit/Ithemal) in the hope that performance engineers and compiler designers can use and improve upon our approach.

Ithemal demonstrates that future systems can leverage data-driven techniques to either augment or fully replace manually developed throughput estimators.

### 2 Motivating Examples

Analytical modeling is difficult for many code sequences; consider examples (a)-(c), their actual throughput\(^1\), and their associated throughput predictions in Table 1. These flawed predictions occur in spite of many hours spent engineering detailed models of underlying microarchitectural details. In contrast, Ithemal’s data driven approach intrinsically learns accurate predictions from the ground truth data.

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>Ithemal</th>
<th>Actual</th>
<th>IACA</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov rax, 0x01</td>
<td>35</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>mov rax, 0x01</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mov rax, 0x01</td>
<td>96</td>
<td>96</td>
<td>96</td>
</tr>
<tr>
<td>mov rax, 0x01</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>mov rax, 0x01</td>
<td>102</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>mov rax, 0x01</td>
<td>83</td>
<td>83</td>
<td>83</td>
</tr>
</tbody>
</table>

\(^1\)Note that – following convention – we define throughput to be the number of clock cycles taken to execute a basic block; this is actually the reciprocal of the standard definition of throughput. We also report throughput for 100 iterations of a given basic block.

**Implementation Errors:** Intel provides extensive documentation of its microarchitectural implementation that enables developers to build performance models for assembly...
Vendor Documentation Errors: The sheer volume of implementation details also means that Intel’s documentation can be incorrect. Tools that faithfully adhere to the documentation can therefore still be incorrect. Sequence (c) is a short sequence with a data dependency that is bypassed within the processor pipeline: the **mov** instruction does not consume many additional clock cycles over that of **shl**. The throughput of this basic block is therefore dominated by the throughput of **shl**. However, the throughput value that Intel provides in its documentation (50 cycles) assumes that there are no dependencies. Therefore, while IACA – Intel’s own tool – closely predicts the value, llvm-mca is incorrect because it uses the dependency-free throughput value. In comparison, Ithemal closely predicts the actual throughput because it works with actual performance data.

3 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 multiscale RNN (Shuai et al., 2015; Zhu et al., 2016) 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.

3.1 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:

\[
\text{mul}, \langle S \rangle, \text{eax}, \text{ecx}, \langle D \rangle, \text{edx}, \text{eax}, \langle E \rangle
\]

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

Assembly code permits more than just register operands, such as constants and memory operands. We map all constants (e.g. integer constants, 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. We present the full canonicalization scheme in Appendix A.
3.2 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) (Hochreiter & Schmidhuber, 1997) cells.

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

3.3 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 (\( h_{\text{block}} \)), Ithemal predicts the basic block’s throughput with a linear layer. Specifically, Ithemal computes \( w \cdot h_{\text{block}} + b \), 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.

The hierarchical combination of the RNN in the instruction layer and the RNN in the prediction layer has several benefits over a non-hierarchical model:

- Instructions are embedded atomically: the prediction layer is only able to generate throughput estimates at specific points in the overall token stream, i.e. after complete instructions. This means that the network does not have an obligation to produce states that correspond to predictions at points in between instructions.

We compare this hierarchical architecture against other architecture choices in Section 6, showing the efficacy of Ithemal’s hierarchical architecture.

4 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.

4.1 Dataset

Table 2 summarizes the set of applications in our 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 as well as end user applications used in day-to-day computing.

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 use Dynamorio (Bruening et al., 2012), a dynamic binary instrumentation tool, to dump the encoded bytes of the executed x86-64 basic blocks. We execute the benchmarks using the standard inputs provided by the benchmark suites. Next, we de-duplicate the dataset by removing basic blocks with same encoded byte patterns. This step is important to eliminate repeated occurrences of basic blocks created by code shared through common header files and by common compilation patterns.

4.2 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 using a script that we have developed that is similar to Agner Fog’s timing script\(^4\). Agner Fog’s timing

\(^4\)https://www.agner.org/optimize/testp.zip
script is commonly used for validating individual instruction throughputs. Our timing script additionally 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. Table 2 shows the breakdown of basic block counts for each benchmark for Haswell microarchitecture in total as well as after de-duplicating repeated basic blocks on a per benchmark basis. The final Haswell dataset, which is de-duplicated across benchmarks, constitutes 1,416,473 unique basic blocks.

<table>
<thead>
<tr>
<th>Benchmark suite</th>
<th>Description</th>
<th>#Total Blocks</th>
<th>#Unique Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Shared Libraries</td>
<td>linux loaders, standard library and other utilities</td>
<td>313846</td>
<td>103977</td>
</tr>
<tr>
<td>SPEC2006 (SPEC, 2006)</td>
<td>benchmark suite with compilers, chess engines, video compression and various simulation applications. Commonly used for benchmarking compilers</td>
<td>247047</td>
<td>141051</td>
</tr>
<tr>
<td>SPEC2017 (SPEC, 2017)</td>
<td>similar to SPEC2006, but with a larger variety</td>
<td>616899</td>
<td>234588</td>
</tr>
<tr>
<td>NAS (NASA, 1991–2014)</td>
<td>benchmarks with stencil computations (dense loops)</td>
<td>3935</td>
<td>1813</td>
</tr>
<tr>
<td>polybench-3.1 (Pouchet, 2012)</td>
<td>polyhedral compilation test suite (dense loops)</td>
<td>1900</td>
<td>859</td>
</tr>
<tr>
<td>TSVC (Maleki et al., 2011)</td>
<td>suite for testing compiler auto-vectorization</td>
<td>5129</td>
<td>2350</td>
</tr>
<tr>
<td>cortextsuite (Venkata et al., 2009)</td>
<td>computer vision workloads including neural networks</td>
<td>6582</td>
<td>3968</td>
</tr>
<tr>
<td>simd (Ihar et al., 2018)</td>
<td>heavily hand vectorized image processing library (exposes lot of SSE2, AVX, AVX2 variants)</td>
<td>212544</td>
<td>25462</td>
</tr>
<tr>
<td>compilers/interpreters</td>
<td>clang (Lattner &amp; Adve, 2004) and different versions of python (2.7,3.5)</td>
<td>2746275</td>
<td>924663</td>
</tr>
<tr>
<td>end user applications</td>
<td>gimp filters, firefox, open-office, rhythmbox, etc.</td>
<td>83555</td>
<td>35513</td>
</tr>
<tr>
<td>Full Dataset</td>
<td></td>
<td>4237712</td>
<td>1416473</td>
</tr>
</tbody>
</table>

Table 2. Composition of the dataset for Haswell, showing the total number of basic blocks per benchmark as well as the unique number of blocks after de-duplicating repeated blocks on a per-benchmark basis. Note that the #Unique Blocks on the full dataset is not equal to the sum of unique blocks per individual benchmark, as we de-duplicate across all benchmarks.

5 Evaluation

We have evaluated Ithemal against two state-of-the-art, hand-written analytical models: IACA (Intel, 2017) (v3.0-28-g1ba2cbb) and llvm-mca (Di Biagio & Davis, 2018) (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 5.1) while maintaining just as fast prediction speeds (Section 5.2). Further, we show that our approach is portable across different microarchitectures in Section 5.3 by showing that Ithemal learns a model that outperforms IACA and llvm-mca without any neural network architecture or hyperparameter modifications.

5.1 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 datasets for each microarchitecture according to the methodology described in Section 4.3.

Table 3 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 several configurations of a basic block is the fastest, and do not calculate each block’s absolute performance.

Figure 2 presents three heatmaps relating actual and predicted values in for basic blocks with throughputs less than 1000 cycles for each prediction method (representing 95% of our dataset).

To generate each heatmap, we binned the actual and predicted data into axis-aligned bins of width and height 20 cycles. The color in each bin represents the count of blocks in that bin. A perfect estimator would have all points along the line $y = x$ (shown as a faint grey, dashed line on the heatmaps), since the predicted throughputs would always match the measured throughputs. We see a higher density near the identity line for Ithemal, compared to both llvm-mca and IACA. Both llvm-mca and IACA also have more horizontal banding, representing more predictions of the same throughput value for different blocks that do actually have different behaviors. Figure 3 shows the average error of each system across a range of throughputs. Compared to llvm-mca and IACA, Ithemal is better for blocks of almost all sizes. All estimators struggle with blocks with measured throughputs just below peaks in the measured throughput distribution. We hypothesize that these blocks correspond to special microarchitectural optimizations that llvm-mca and IACA do not model. Ithemal also struggles some with these rare blocks, but still outperforms both analytical estimators.

### 5.2 Speed

Table 4 presents the results of our evaluation of estimator throughput: the number of instructions able to be timed per second for each estimator. We calculate this by measuring the number of basic blocks each tool can time per second, and multiplying that by the average number of instructions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ivy Bridge</td>
<td>llvm-mca</td>
<td>0.181</td>
<td>0.902</td>
<td>0.777</td>
</tr>
<tr>
<td></td>
<td>Ithemal</td>
<td>0.089</td>
<td>0.955</td>
<td>0.913</td>
</tr>
<tr>
<td>Haswell</td>
<td>llvm-mca</td>
<td>0.200</td>
<td>0.890</td>
<td>0.790</td>
</tr>
<tr>
<td></td>
<td>IACA</td>
<td>0.209</td>
<td>0.917</td>
<td>0.833</td>
</tr>
<tr>
<td></td>
<td>Ithemal</td>
<td>0.089</td>
<td>0.960</td>
<td>0.918</td>
</tr>
<tr>
<td>Skylake</td>
<td>llvm-mca</td>
<td>0.239</td>
<td>0.852</td>
<td>0.729</td>
</tr>
<tr>
<td></td>
<td>IACA</td>
<td>0.167</td>
<td>0.926</td>
<td>0.835</td>
</tr>
<tr>
<td></td>
<td>Ithemal</td>
<td>0.079</td>
<td>0.960</td>
<td>0.895</td>
</tr>
</tbody>
</table>
per basic block. In the last row, we also show the corresponding estimator throughput if we instead measure the ground-truth throughput for a given block. We measured these estimator throughputs on the Haswell test set on a machine with an Intel Xeon E5-2680 CPU.

Ithemal is as fast as llvm-mca and IACA in our measurements, and is significantly faster than empirical evaluation of basic blocks. It is worth noting that llvm-mca and IACA can both also output diagnostic information about basic blocks, and also that empirical evaluation of ground-truth data could be sped up by running fewer repeated measurements (it may be possible that as few as 2 or 3 measurements would suffice in some contexts). However even with these qualifications, we show that 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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Throughput (Instructions / second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>llvm-mca</td>
<td>492</td>
</tr>
<tr>
<td>IACA</td>
<td>541</td>
</tr>
<tr>
<td>Ithemal</td>
<td>560</td>
</tr>
<tr>
<td>Empirical execution</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 4. Estimation throughputs for different estimators measured in instructions per second

5.3 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 3 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.

6 Neural Network Architecture Exploration

We evaluated a number of neural network architectures with varying levels of structure and complexity before arriving at Ithemal’s network architecture (Section 3).

Figure 4 shows a DAG-RNN (Shuai et al., 2015; Zhu et al., 2016). Instructions are embedded identically as to in Ithemal (i.e. the token layer and instruction layer remain the same). However rather than running an RNN sequentially over the instructions in the prediction layer, the DAG-RNN constructs a dependence graph of the instructions in the basic block, with a directed edge between a pair of instructions if the second instruction depends on an output of the first instruction. Because an instruction can depend on multiple previous instructions, we apply an element-wise max to reduce the states feeding in to a given instruction. Then, the DAG-RNN applies an LSTM cell, using the generated instruction embedding as the input, and the result of the element-wise max as the input state. To generate the final prediction, we take an element-wise max of the output states of all leaf instructions (all instructions with no dependents), and pass the result through a linear layer.

The DAG-RNN is inspired by the theoretical behavior of a perfect out-of-order processor: the throughput of a basic block running on a perfect out-of-order processor is equivalent to the throughput of the longest path that must be serially executed in that basic block. Using a DAG-RNN implicitly encodes this prior, by only allowing information to propagate through the paths that must be serially executed in the block.

We also tested a simple token-level RNN with LSTM cells, which has a similar base architecture as Ithemal, but without the topmost prediction layer. Instead, this model sequentially consumes all tokens in a basic block making no explicit distinction between instructions, giving a baseline measure for the efficacy of Ithemal’s hierarchical model. The full architecture diagram for the token-level RNN is shown in Appendix E.
Results. Figure 5 shows the training and validation loss for each model across the first five epochs. It is clear that the hierarchical LSTM is the best model among these three. The sequential LSTM performs the worst by far, motivating the need to process individual tokens and instructions at multiple scales. The fact that the DAG-RNN performs worse than the hierarchical LSTM implies that the exact ordering of instructions in a basic block does matter, not just the dependency chains. This aligns with the fact that instruction scheduling optimizations in compilers do result in changed performance, despite the underlying dependency graph being the same. While the perfect out-of-order execution model is a reasonable approximation, modern processors do in fact have some serial behavior, which a sequential model is able to capture.

7 Related Work

Hierarchical Multiscale RNNs Hierarchical RNNs are a classic technique in sequence prediction domains (El Hihi & Bengio, 1995). Ithemal uses a hierarchical multiscale RNN as the core of its prediction methodology, similar to many other proposed hierarchical RNN models. Ithemal has some inherent advantages over other models from the literature: as opposed to the methodology proposed in (Chung et al., 2017), the hierarchical structure in instruction embedding and throughput prediction is explicit rather than latent. Ithemal’s structure is instead most similar to boundary-aware hierarchical RNNs, such as the one presented in (Baraldi et al., 2017) for video captioning based on a hierarchy of frames and scenes.

DAG-RNNs and Graph Neural Networks Neural networks with generic graph based structures have been used in NLP tasks to model relations among words in sentences (Peng et al., 2017; Dhingra et al., 2017). Programs also can be represented using Gated Graph Neural Networks (Alamanis et al., 2018), to perform high-level tasks such as variable naming, or identifying variable misuses. Xu et al. (2017) uses graph neural networks to find binary similarity between different execution platforms. In experimenting with a DAG-RNN (Shuai et al., 2015; Zhu et al., 2016), we hoped to be able to take advantage of the program semantics of a basic block, although that approach does not perform as well in our collected datasets.

Analytical Models for Throughput and Runtime Estimation Apart from state-of-the-art tools like Illvm-mca and IACA, other analytical models exist for throughput estimation (Taha & Wills, 2003) of instructions. OS-ACA (Laukemann et al., 2018) is an open source analytical model similar to Illvm-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 (Chen & Aamodt, 2009) to estimate throughput for multithreaded programs. Cycle-accurate simulators such as ZSim (Sanchez & Kozyrakis, 2013) and Marss (Patel et al., 2011) have a high start-up cost and are more suited for coarse grained simulations.

Coarser analytical models exist for predicting program runtimes (Park, 1993). Work has also been done in developing analytical models to predict performance of restricted classes of programs. For example, work on predicting parallel program runtimes include (Rugina & Schauser, 1998; Adve & Vernon, 2004; Hartleb & Mertsiotakis, 1992; Blanco et al., 2004; Fahringer & Zima, 1993) and work on predicting worst case execution times include (Ferdinand et al., 2001; Li et al., 2007).

All of these models require detailed processor modeling and considerable human development effort.

Learned Models for Throughput and Runtime Estimation There has been work on developing machine learning–based models for absolute and relative runtime estimation. (Huang et al., 2010) introduces sparse polynomial regression to predict execution time of programs by using a set of hand–crafted features of high level programs. Dubach et al. (2007) uses neural networks with hand-crafted features to estimate the speedup between two code sequences. GameTime (Seshia & Kotker, 2011; Seshia & Rakhlin, 2012) 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 architectural knowledge embedded into the model.

Microarchitectural Predictions Similar to basic block throughput estimation, various microarchitectural prediction tasks have been explored with machine learning. For example, RNN models can be used for predicting memory access (Hashemi et al., 2018), and perceptron models can be used for branch prediction (Jimenez & Lin, 2001).

8 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, while minimizing developer effort.
Acknowledgments

We would like to thank Yishen Chen, Ajay Brahmakshatriya for their help in creating the datasets, and all reviewers for insightful comments and suggestions. We would also like to thank Ondřej Sýkora and Jon Orwant for their many helpful discussions. This research was supported by DARPA D3M Award #FA8750-17-2-0126, NSF Grant CCF-1751011, DARPA/Nvidia Symphony Award #HR0011-18-3-0007 and DARPA Award #HR001118C0059. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References


Ithemal: Basic Block Throughput Prediction


A Canonicalization Scheme

We demarcate memory operands (consisting of a base address, and an optional offset and displacement) by surrounding them with \(<M>\) and \(</M>\) delimiter tokens.

The following is the full grammar for the token strings, as described in Section 3.1.

\[
\begin{align*}
\text{<block>} & \::= \text{<instr>} + \\
\text{<instr>} & \::= \text{opcode} \ <S> \ <opnd>* \ <D> \ <opnd>* \ <E> \\
\text{<opnd>} & \::= \text{register} | \ <M> \ \text{register} + \ </M> | \text{CONST}
\end{align*}
\]

B Training Hyperparameters

Here we present the hyperparameters for the model as described in Section 3. All vectors, including the embedding width, hidden, and output states have width 256. We train our models using asynchronous SGD, with a batch size of 4, and 6 parallel trainers. The initial learning rate is 0.1, and after the first 2 epochs, it decreases by a geometric factor of 1.2 every epoch. We use the default PyTorch formulation for momentum (i.e. not Nesterov momentum) with \(\beta = 0.9\). Each parallel trainer samples without replacement from the dataset until all training data is exhausted. If a trainer hits a \(\text{NaN}\) gradient, that trainer is halted for the remainder of the epoch, and the elements in that trainer’s batch are dropped for the epoch. At the beginning of the next epoch, all trainers are restarted. Training halts once all trainers are halted and an epoch cannot be completed.

C Heatmaps of Different Prediction Methods

Figure 6 shows all prediction heatmaps for Ithemal, llvm-mca and IACA under the Intel Ivy Bridge, Haswell and Skylake microarchitectures. Note that the latest IACA version does not support Ivy Bridge and hence its prediction heatmap is not available.

D Prediction Errors for Throughput Ranges

Figure 7 shows how the average error changes between various throughput ranges for each prediction method under different microarchitectures for basic blocks with throughput values under 1000 cycles. Throughput values are broken up in to bins of length and width 20 cycles on each axis. It also shows the throughput distribution of the basic blocks, and the average error across different measured throughput ranges. Ithemal consistently predicts throughput values with lower average errors compared to llvm-mca and IACA. Overall, Ithemal is more robust in its prediction across all throughput ranges compared to llvm-mca and IACA which show higher fluctuations.

E Token RNN Architecture

The full architecture for the Token RNN as presented in Section 6 is shown in Figure 8.
Figure 6. Heatmaps for measured and predicted throughput values under different models for basic blocks with measured throughput values less than 1000 cycles for the Intel Ivy Bridge, Haswell and Skylake microarchitectures.
Ithemal: Basic Block Throughput Prediction

Figure 7. Average error curves and throughput distributions for different models for basic blocks with measured throughput values less than 1000 cycles under different microarchitectures for the test set.
Figure 8. Token RNN Architecture