A Learnt IR level Precise Throughput Cost Model using Graph Neural Network with Interference Graph

Zhenyan Zhu, Yuxuan Xia, Arshdeep Singh, Zhan Shi
University of Michigan AnnArbor, MI
{lukezhuz, xyuxuan, arshdp, zhanshi}@umich.edu

***Abstract*— This research paper delves into the problems associated with developing Intermediate Representation (IR) level basic block throughput measurement models with graph neural network. Two primary challenges are identified: Firstly, the absence of a representative data-set mapping from IR level basic blocks to hardware measurements. Secondly, the dependency of the IR level cost model on subsequent compiler transformations and optimizations, which translate the provided IR into target assembly code. To address these challenges, we introduce a novel methodology for collecting inverse throughput of IR level basic blocks. Additionally, we propose an innovative approach that involves integrating interference edges into graph neural networks. This method is designed to enhance the IR cost model's ability to bridge the gap between an unlimited number of variables and the constraints imposed by limited physical registers. Our approach represents a significant step forward in the field of building efficient and accurate IR level cost model that can be applied in any compiler optimizations requires to query IR level cost.**

***Index Terms*— LLVM, MIR, Inverse Throughput, Graph Neural Network, Interference Graph**

I. INTRODUCTION

Machine learning these days has been playing an important role in guiding Compiler Optimization [7]. In machine learning-based compiler optimization, it's crucial to determine the cost of basic blocks/whole programs. However, querying the cost by running the code on real hardware is too expensive and time-consuming. In addition, Many current analytical models, like llvm-mca [1] and ICIA [2] , fail to capture the exact mechanisms of the processors as underlying microarchitectures are extremely complex and changing frequently.

To alleviate this issue, Deep Neural Networks based performance model, Ithemal[3] was born. It utilizes sequential Long-Short Term Memory (LSTM) to learn a representation of basic blocks followed by a linear transformation to predict the throughput values, which has less than half the error of llvm-mca and ICIA. However, Ithemal does not fully capture data dependency in basic blocks and has to train the entire model separately on different microarchitectures. A new model, Granite[4], improved upon Ithemal. It utilized the graph neural network to capture the relational information in the graph, and it applied multi-task learning[5] to reduce training cost. Granite reduced the error by 1.7% while improving training and inference throughput by approximately 3.0x[4] compared to Ithemal.

However, to guide IR-level compiler optimization decisions, a machine code level cost model may not be so helpful since it is still required to transform the IR to target assembly by running the whole compiler backend. Hence, we need an IR-level cost model that can accurately and efficiently produce cost given a sequence of IR to assist compiler optimization algorithms explore more optimization spaces and hopefully find the optimal transformation.

One of the challenges presented in IR-level cost models is that the learnt cost model have no knowledge of how the provided IR is being further lowered and optimized. This issue arises because the model typically receives only the input IR and the hardware-measured inverse throughput, without additional contextual information. Specifically in this paper, we focus on a challenge posed by the register allocation pass as the model does not know whether a variable will be presented allocated on a register or memory. To bridge this gap, we propose an innovative approach by incorporating interference edges into the graph representation of basic blocks in order to let the graph neural network predict and account for the complexities introduced during the register allocation phase.

II. MOTIVATING EXAMPLE

Here is a illustrative example on how register allocation can impose a discrepancy in generated target assembly as well as on hardware measurement given the same IR-level basic block. Given the same IR-level basic block

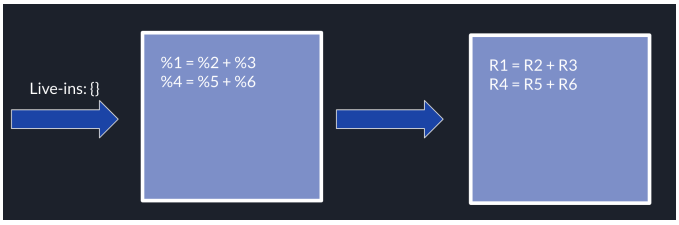

Fig. 1. one possible register allocation transformation with low register pressure

with different register pressure in context, it's more likely that the blocks with high register pressure will have more

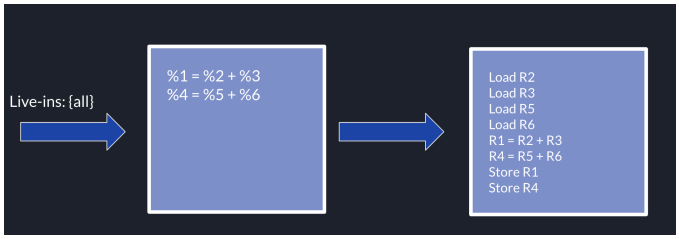

Fig. 2. one possible register allocation transformation with high register pressure

variables allocated on memory, and memory operations may greatly enhance the overall inverse throughput.

III. HIGH-LEVEL ARCHITECTURE

The presented IR level cost model is an extension of Google's gematria repository[4]. It is composed of the following building blocks as illustrated in figure 3.

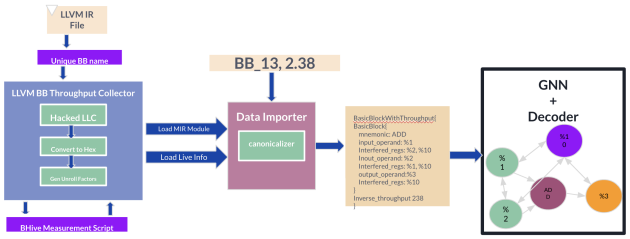

Fig. 3. High-level Architecture of Data Collector and Learnt IR cost model

- **LLVM IR basic block throughput collector** -> The first step is to obtain the hardware measured inverse throughput of each IR-level basic block as the model's ground truth. We encountered the challenge that the code generator may split a IR basic blocks into several assembly level basic blocks. By hacking into `llc`, we can group those splited basic blocks together in assembly printing pass as well as filtered out special basic blocks or instructions that have special impact.
- **Data Importer** -> Next, hardware measurements, Machine IRs that dumped before register allocation and Live Intervals dumped from `llvm::LiveIntervals` analysis pass will be imported by the Data Importer and transformed into canonicalized protobuf objects as the training or testing data-set
- **Graph Builder** -> After a representative training data-set and a testing data-set are prepared, each basic block within those data-sets will be transformed into a graph representation by constructing input, output, structural dependency as well as interference dependency as edges and register, virtual register, opcodes as nodes.
- **Graph Neural Network & Decoder Network** -> Finally, the graph representation as well as the hardware measurement will be passed to a graph neural network connected with a decoder network. The parameters within those networks will be updated during training epochs based on the constructed graph representation and provided ground truth.

IV. A LLVM IR BASIC BLOCK THROUGHPUT COLLECTOR

The LLVM IR Basic Block Throughput Collector takes an LLVM IR file as the input and outputs inverse throughput of each measured-able basic blocks within the given file. It contains the below parts:

1. throughput measurement of assembly-level basic block:
To measure the inverse throughput of a assembly-level basic block, we utilized the existing benchmark tool: BHive[6]. BHive provides us with a measurement script that takes a hex representing an X86 basic block and a unroll factor α. It will unroll the provided code for α times and measure the total inverse throughput(clock cycles) of the sum of measurement script and unrolled code execution. Then, to calculate the inverse throughput of the assembly-level basic block, we extend BHive's script to obtain the inverse throughput of one single basic block using the following formula:

$$throughput(b) = \frac{cycles(b,\alpha) - cycles(b,\beta)}{\alpha - \beta}$$

where α and β are two unroll factors correspondingly, $\alpha > \beta$, $cycles(b,\alpha)$ is the total clock cycles of unrolling basic block b for α times.

2. maintaining the mapping from IR-level basic block to assembly-level basic block:
Considering the potential for merging or splitting of IR-level basic blocks by the LLVM compiler (`llc`) during code generation phases, the next step involves establishing a correlation between IR-level and assembly-level basic blocks. Upon an in-depth exploration of the functionalities offered by `Machine Basic Block`, it has been found that the `getBasicBlock()` method is capable of retrieving the corresponding LLVM IR basic block. Then, we hacked the `X86AsmPrinter` pass in `llc` to group the separated assembly basic blocks together. We filtered out control instructions because inverse throughput is measured by unrolling basic blocks. In addition, basic blocks that starts a function or ends a function will also be filtered as they impose additional overhead of function prologue and epilogue.

3. put everything together:
The LLVM IR Basic Block Throughput Collector processes an LLVM IR file initially by instrumenting each basic block with a unique name. This preparatory step is crucial for traceability throughout the process. Following this, it is utilizing the hacked `llc` as mentioned above to group assembly-level basic blocks according to their corresponding IR-level unique name. This grouping is essential for maintaining accurate mapping despite the potential alterations made during the compilation process, such as merging or splitting of basic blocks.

Subsequently, the system converts these grouped basic blocks into hexadecimal code representations, carefully determining unroll factors that are proportional to the code size.

Finally, the extended BHive script is applied to measure the inverse throughput of each IR-level basic block. A high-level workflow can be found in figure 4.

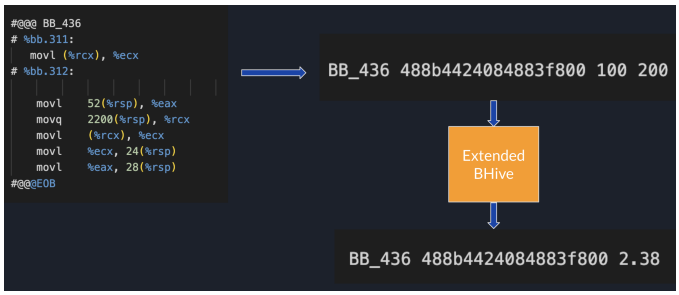

Fig. 4. transformations happening during the data collection phase

V. Data Importer: BHiveImporter

A. Workflow of Data Importer

The input of the Data Importer are three files: the MIR file (intermediate representation between llvm IR and target assembly) that dumps before register allocation pass, the registers' live range file, which contains the live range of used virtual and physical registers in each function extracted from `llvm::LiveIntervals`, and the performance measurement file that maps from each basic block unique name to hardware measurement collected by the LLVM IR basic block throughput collector.

The BHiveImporter will first load the MIR module using `llvm::MIRParser` to obtain the mapping from unique name to each machine basic block data called `name_to_MBB`, and it will also parse the live range information file and construct a mapping from each unique function name to the live range of each registers.

Next, the BHiveImporter will iterate over each line of the performance measurement file containing the unique basic block name as well as the measured inverse throughput. The `llvm::MachineBasicBlock` is queried from `name_to_MBB` using the unique name as the key, and its canonicalized instructions containing corresponding opcode, input and output operands are extracted and stored into the protobuf representation for serialization.

Finally, for each input/output registers, the interfered register list will be constructed by calculating the intersection of live ranges recorded as shown below.

B. Intersection Algorithm

Upon request, the BHiveImporter should return the interference graph specifying interfered virtual and physical registers. We could output per-basic-block interference relationship and per-function relationship. The algorithm 1 is used to check whether two registers have colliding live range in a specified basic block. This algorithm is used during graph building phase. The graph builder takes in a .perf performance file, which contains each basic block's name as well as its inverse throughput. It will query the BHiveImporter for interference information of each basic block. Part B and C describes necessary algorithms used to obtain such information.

Since a register could have multiple live ranges, the input would be two lists: $reg\_live\_interval1$ and $reg\_live\_interval2$, that specify the live ranges of each registers. There is additional bb_range argument that denotes in which basic block we expect the collision to appear. The function will only return true if there is a collision between any live ranges between two registers, and this collision intersects with the basic block specified by bb_range. This is what we called per-basic-block interference relationship. If we ignore the second condition, this is what we call per-function interference.

Algorithm 1: LiveRange Intersection Algorithm

```
1:  Function checkRegIntersectionsWithBBRange(
2:  reg_live_interval1
3:  reg_live_interval2
4:  bb_range) is
5:      foreach interval: reg_live_interval1 do
6:          Check == areIntersected(interval, bb_range) ;
7:          if Check then
8:              Indicate reg1 do intersect bb_range ;
9:              record intersection range as RANGE;
10:     if !Check then
11:         return false ;
12:     foreach interval: reg_live_interval2 do
13:         if areIntersected(interval, bb_range) &&
14:         areIntersected(interval, RANGE) then
15:             return true ;
16:     return false ;
17: Function areIntersected(range1, range2) is
18:     if range1.second <= range2.first ||
19:     range2.second <= range1.first then
20:         return false ;
21:     return true ;
```

C. Add interference Graph

With live range intersection algorithm, our BHiveImporter could update protocol buffer with interference information. The *addInterferenceGraph* function will first initialize two maps: live_virtual_registers and live_physical_registers to hold live information of registers in the current basic block. These two maps will be used when we search collision of live ranges given a register used.

Additionally, this function uses three helper functions: The update_live_regs helper function will update the pre-

vious two maps using the provided operand as the argument. The operand is a $CanonicalizedOperandProto$ data structure that stores properties of the register. The add_interference_on_name helper function is used to add interference relations given a register name. It will check interference of given register with all registers in live_virtual_registers and live_physical_registers. If such interfered register is found, it will be added to the given register's canonicalized protocol buffer data structure. The add_interference helper function adds interference information of the given register using add_interference_on_name helper function.

The overall workflow of *addInterferenceGraph* function will be as follows: it first iterate through all canonicalized instruction of the current basic block protocol buffer. Then it uses update_live_regs helper function on every input and output operands of the instruction to obtain live range information. Then it will iterate through mutable operands of all mutable canonicalized instructions and use add_interference helper function to add its interference. The function will return true after doing all the work above.

VI. Model Architecture Details

A. Graph Encoding of Basic Blocks

We inherit the graph representation of basic blocks from GRANITE [4], with several modifications to account for the semantic gap between X86 assembly and LLVM MIR. In particular, our change is as follows:

- The "Register" value node type now contains virtual registers, in addition to physical registers. Virtual registers are represented in token by their type ("VREG32", "VREG64", etc.)
- We added a new edge type, "Register Interference", to denote interference relation between virtual registers. The interference graph comes from the data importer.

The following tables I and II show the node types and edge types after our modification [4]:

Node Type	Token
Instruction Nodes	
Mnemonic	The mnemonic of the instruction (e.g. ADD).
Prefix	The prefix of an instruction (e.g. LOCK).
Value Nodes	
Register	Register name (e.g. RBX) or virtual register type (e.g. VREG64).
FP immediate value	Special token shared by all floatingpoint immediate values.
Immediate value	Special token shared by all immediate value nodes.
Address computation	Special token shared by all address computation nodes.
Memory value	Special token shared by all values stored in memory.

TABLE I

THE NODE TYPES IN GRANLTE GRAPH REPRESENTATION, MODIFIED FROM [4].

Edge Type	Description
Structural Dependency	From an instruction mnemonic node to the instruction mnemonic node of the following instruction.
Input Operand	From a value node to an instruction mnemonic node.
Output Operand	From an instruction mnemonic node to a register or a memory value node.
Address Base	From a register node to an address computation node.
Address Index	From a register node to an address computation node.
Address Segment	From a register node to an address computation node.
Address Displacement	From an immediate value node to an address computation node.
Register Interference	Between interfered registers. Each pair of interference would have 2 edges, one from each to the other.

TABLE II

THE EDGE TYPES IN GRANLTE GRAPH REPRESENTATION, MODIFIED FROM [4].

In our experiments, we trained three different models with three different levels of interference graphs information: One with no interference graph (i.e. the "Register Interference" edge type is never used), one with interference graph in a per-basic block level, and one with interference graph in a per-function level.

B. Graph Neural Network

The graph neural network (GNN) consists of an encoder and a decoder network at its core. The encoder is used to learn the embedding for the nodes, edges and the graph at a basic block level. This is an iterative process and after user-defined number of iterations, the encoder is able to learn the embedding for each of the components of a basic block. The decoder is a feed forward neural network with ReLU activation function at its nodes. The decoder consumes the encoder output and learn the parameters to predict the inverse throughput of the basic block.

The nodes, edges and the graph level embedding vectors are initialized each with a vector size of 128. We extended the granite's GNN model to support the virtual registers and the interference edge types to add additional relevant information into the model. We built 3 different models for out project i.e model with no live information, model with live information per basic block and a model with live information at a global (function) level. Each model is trained on approximately 12,000 basic blocks and tested on the 768 basic blocks comprising of various benchmarks.

The loss function for the model is defined as the mean absolute error (MAE) which quantifies the difference in the actual and the predicted inverse throughput. Large difference in the MAE means large loss and the objective of the model is to minimize the loss function thereby increasing the prediction accuracy.

VII. Results Analysis

We trained the model for 5000 epochs and a learning rate of 0.001. A mini-batch gradient descent algorithm is used to learn the model and the Figure 5 shows our model was able to train, learn and reduce the loss with number of epochs for all three different model types.

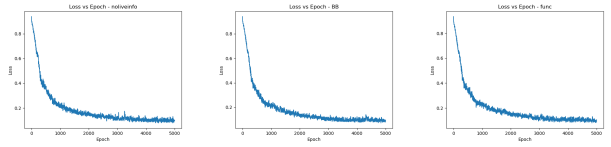

Fig. 5. Training loss convergence

The performance of the model is evaluated using the testing dataset in terms of mean absolute error and mean squared error (MSE) as shown in Figure 6.

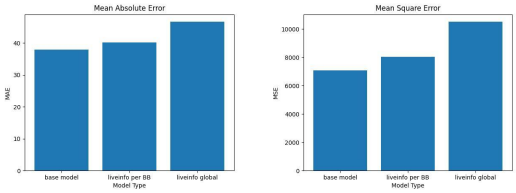

Fig. 6. MAE and MSE performance for three different models

Further, we analyse the MAE on the different benchmarks carved out carefully from the testing dataset as shown in figure 7

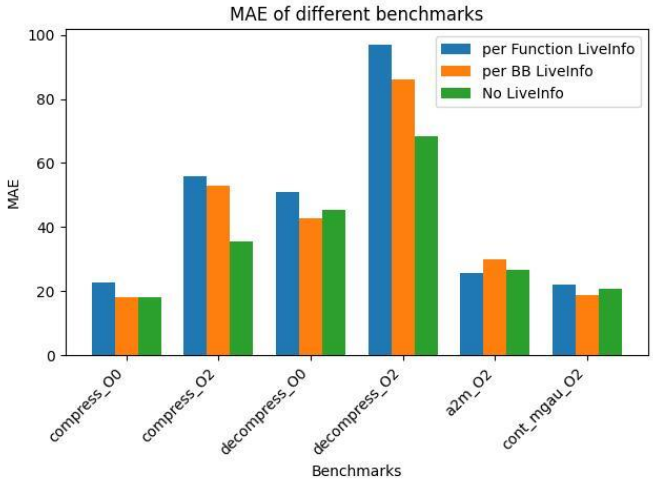

Fig. 7. MAE performance for three different models across various benchmarks

Generally, it's anticipated that a model trained with more information in the data would outperform a simpler version. Therefore, it's logical to assume that the model integrating both global and basic block level live information would surpass the base model's performance. This would typically result in lower MAE and MSE for the former model with richer information. However, the lack of significant improvement in this case can be attributed to the limited data available. Our current model is more intricate than the previous 'granite' model, which was developed using 1.4 million basic blocks. We are confident that the performance of our model will greatly improve once it is reinforced with a more substantial dataset.

VIII. Related Work

A. ML-driven Hardware Cost Model for MLIR

There has been some recent work on cost modelling of high-level programs or Intermediate Representations. One of the most related and latest IR level cost model is a ML-driven hardware cost model by Intel[9]. This model is first of its kind which tackles hardware performance or bottleneck predictions for MLIR code. The model is based on Conv1D and Maxpool, which makes inference extremely fast and accurate compared to the likes of LSTM or even GNNs. However, this work is not open-source and we have no access to evaluate the models.

IX. Conclusions

In this paper, we presented a comprehensive approach to developing an efficient and accurate IR-level basic block throughput measurement model. Our work addresses critical gaps in existing methods, offering innovative solutions that cater to the evolving needs of compiler optimization and performance analysis.

Our LLVM IR Basic Block Throughput Collector, an integral component of this research, demonstrates the ability to accurately and efficiently collect hardware-measured inverse throughput of IR-level basic blocks. By instrumenting basic blocks with unique identifiers and manipulating the `llc` compiler to group assembly-level basic blocks, we ensure precise mapping and data integrity throughout the process. The extension of the BHive script further enables detailed and accurate throughput measurements.

Furthermore, the integration of interference edges into the graph representation of basic blocks represents a significant modeling advancement. However, the enhanced model currently does not outperform the basic version, likely due to limited data representation and increased complexity. Our future efforts will focus on using ComPile [8], a recent comprehensive dataset, for training, which we expect will improve the model's performance and fully harness the potential of this innovative approach.

Acknowledgment

We would like to thank Ondřej Sýkora from Google Research for providing valuable comments and advice on this project as well as helping us understand the `gematria` repository.

References

[1] llvm-mca- llvm machine code analyzer, `https://llvm.org/docs/CommandGuide/llvm-mca.html`

[2] Intel architecture code analyzer (2017), `https://software.intel.com/en-us/articles/intel-architecture-code-analyzer,2017`

[3] C. Mendis, A. Renda, D. Amarasinghe, and M. Carbin, (2019). Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. Avaliable: https://proceedings.mlr.press/v97/mendis19a/mendis19a.pdf

[4] O. Sykora, P. Phothilimthana, C. Mendis, A. Yazdanbakhsh (2022). GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. Available: https://arxiv.org/abs/2210.03894.

[5] R. Caruana (1997). Multitask Learning. Available: https://link.springer.com/article/10.1023/A:1007379606734

[6] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. S'ykora, S. Amarasinghe, and M. Carbin (2019). BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. Avaliable: https://ieeexplore.ieee.org/document/9042166

[7] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. ACM Comput. Surv. 51, 5, Article 96 (September 2019), 42 pages. Avaliable: https://doi.org/10.1145/3197978

[8] Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, Johannes Doerfert (2023). ComPile: A Large IR Dataset from Production Sources. Avaliable: https://arxiv.org/abs/2309.15432

[9] Dibyendu Das, Sandya Mannarswamy (2023). ML-driven Hardware Cost Model for MLIR. Avaliable: https://arxiv.org/abs/2302.11405