# IR2Vec: LLVM IR Based Scalable Program Embeddings

S. VENKATAKEERTHY, ROHIT AGGARWAL, SHALINI JAIN,
MAUNENDRA SANKAR DESARKAR, and RAMAKRISHNA UPADRASTA,
Indian Institute of Technology Hyderabad
Y. N. SRIKANT, Indian Institute of Science

We propose IR2Vec, a Concise and Scalable encoding infrastructure to represent programs as a distributed embedding in continuous space. This distributed embedding is obtained by combining representation learning methods with flow information to capture the syntax as well as the semantics of the input programs. As our infrastructure is based on the Intermediate Representation (IR) of the source code, obtained embeddings are both language and machine independent. The entities of the IR are modeled as relationships, and their representations are learned to form a *seed embedding vocabulary*. Using this infrastructure, we propose two incremental encodings: *Symbolic* and *Flow-Aware*. *Symbolic* encodings are obtained from the *seed embedding vocabulary*, and *Flow-Aware* encodings are obtained by augmenting the *Symbolic* encodings with the flow information.

We show the effectiveness of our methodology on two optimization tasks (Heterogeneous device mapping and Thread coarsening). Our way of representing the programs enables us to use non-sequential models resulting in orders of magnitude of faster training time. Both the encodings generated by IR2Vec outperform the existing methods in both the tasks, even while using *simple* machine learning models. In particular, our results improve or match the state-of-the-art speedup in 11/14 benchmark-suites in the device mapping task across two platforms and 53/68 benchmarks in the thread coarsening task across four different platforms. When compared to the other methods, our embeddings are *more scalable*, *is non-data-hungry*, and *has better Out-Of-Vocabulary (OOV) characteristics*.

CCS Concepts: • **Software and its engineering → Compilers**; **General programming languages**; • **Computing methodologies → Machine learning**; *Knowledge representation and reasoning*;

Additional Key Words and Phrases: LLVM, intermediate representations, representation learning, compiler optimizations, heterogeneous systems

**ACM Reference format:**
S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2Vec: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (November 2020), 27 pages.
https://doi.org/10.1145/3418463

This research is funded by the Department of Electronics & Information Technology and the Ministry of Communications & Information Technology, Government of India. This work is partially supported by a Visvesvaraya PhD Scheme under the MEITY, GoI (PhD-MLA/04(02)/2015-16), an NSM research grant (MeitY/R&D/HPC/2(1)/2014), a Visvesvaraya Young Faculty Research Fellowship from MeitY, and a faculty research grant from AMD.
Authors' addresses: S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, and R. Upadrasta, Dept. of CSE, Indian Institute of Technology Hyderabad, Sangareddy, Telangana, India 502 285; emails: {cs17m20p100001, cs18mtech11030, cs15resch11010}@iith.ac.in, {maunendra, ramakrishna}@cse.iith.ac.in; Y. N. Srikant, Indian Institute of Science, Bangalore, Karnataka, India 560 012; email: srikant@iisc.ac.in.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2020 Copyright held by the owner/author(s).
1544-3566/2020/11-ART32
https://doi.org/10.1145/3418463

## 1 INTRODUCTION

With the growth of computing, comes the growth in computations. These computations are necessarily the implementation of well-defined algorithms [20] as programs. Running these programs on the rapidly evolving diverse architectures poses a challenge for compilers (and optimizations) for exploiting the best *performance*. Because most of the compiler optimizations are NP-Complete or undecidable [44, 48], most of the modern compilers use carefully handwritten heuristics for extracting superior performance of these programs on various architectures.

Several attempts have been made to improve the optimization decisions by using machine learning algorithms instead of relying on sub-optimal heuristics. Some of such works include prediction of unroll factors [53], inlining decisions [54], determining thread coarsening factor [41], Device mapping [26], vectorization [27, 45], and the like. For such optimization applications, it is crucial to extract information from programs so that it can be used to feed the machine learning models to drive the optimization decisions. The extracted information should be in a form that is *amenable to learning* so as to improve the optimizations on the input programs.

Primarily, there are two ways of representing programs as inputs to such machine learning algorithms—Feature-based representations and Distributed representations. Feature-based representation involves representing programs using hand-picked features—designed by domain experts [23, 26, 41]—specific for the particular downstream applications. Examples for the features could be the number of basic blocks, number of branches, number of loops, and even the derived/advanced features like arithmetic intensity. On the other hand, representation learning involves using a machine learning model to *automatically learn* to represent the input as a distributed vector [10]. This learned representation—encoding of the program—is often called *program embedding*. Such a distributed representation is a real-valued vector whose dimensions cannot be distinctly labeled, as opposed to that of feature-based representations.

However, most of the existing works on using distributed learning methods to represent programs use some form of Natural Language Processing for modeling; all of them exploit the statistical properties of the code and adhere to the *Naturalness hypothesis* [3]. These works primarily use Word2Vec methods like skip-gram and CBOW [39], or encoder-decoder models like Seq2Seq [56] to encode programs as distributed vectors.

It can be noted that most of the existing representations of programs have been designed for software-engineering-based applications. This includes algorithm classification [42, 52], code search and recommendation [14, 19, 34, 38], code synthesis [50], Bug detection [59], Code summarization [31], and Software maintenance [1, 2, 6, 25]. We, however, believe that a carefully designed *embedding* that can *encode* the semantic characteristics of the program can be highly useful in making optimization decisions, in addition to being applied for software engineering.

In this article, we propose IR2Vec, an agglomerative approach for constructing a continuous, distributed vector to represent source code at different (and increasing) levels of IR hierarchy—Instruction, Function, and Program. The vectors that are formed lower down the (program abstraction) hierarchy is used to build the vectors at higher levels.

We make use of the LLVM compiler infrastructure [35] to process and analyze the code. The input program is converted to LLVM-Intermediate Representation (LLVM-IR), a language- and machine-independent format. The initial vector representations of the (entities of the) IR, called *seed embeddings*, is learned by considering its statistical properties in a Representation Learning framework. Using these learned seed embeddings, hierarchical vectors for the new programs are formed. To represent Instruction vectors, we propose two flavors of encodings: *Symbolic* and *Flow-Aware*. The *Symbolic* encodings are generated directly from the learned representations. When augmented with the flow analyses information, the Symbolic encodings become *Flow-Aware*.

We show that the generic embeddings of IR2Vec provide superior results when compared to the previous works like DeepTune [21], Magni et al. [41], and Grewe et al. [26] that were designed to solve specific tasks. We also compare IR2Vec with NCC by Ben-Nun et al. [13]; both have a similar motivation in generating generic embeddings using LLVM IR, though using different methodologies/techniques.

We demonstrate the effectiveness of the obtained encodings by answering the following **R**esearch **Q**uestions (RQ's) in the later sections:

***RQ1: How well do the seed embeddings capture the semantics of the entities in LLVM IR?***
As the seed embeddings play a significant role in forming embeddings at higher levels of Program abstraction, it is of paramount importance that they capture the semantic meaning of the entities to differentiate between different programs. We show the effectiveness of the obtained seed embeddings in Section 5.1.

***RQ2: How good are the obtained embeddings for solving diverse compiler optimization applications?***
We show the richness of our embeddings by applying it for different tasks: (a) Heterogeneous device mapping and (b) Prediction of thread coarsening factor in Sections 5.2 and 5.3, respectively.

***RQ3: How scalable is our proposed methodology when compared to other methods?***
We discuss various aspects by which our encoding is more scalable than the others in Section 6. We show that IR2Vec has improved training time, and is non-data-hungry. Also, IR2Vec does not encounter *Out Of Vocabulary (OOV)* words. These are the words that have not been exposed during the training phase, and hence are not part of the seed embedding vocabulary, but are encountered during test/inference phase.

*Contributions*: The following are our contributions:

- We propose a unique way to map LLVM-IR entities to real-valued distributed embeddings, called a *seed embedding vocabulary.*
- Using the above seed embedding vocabulary, we propose a Concise and Scalable encoding infrastructure to represent programs as vectors.
- We propose two embeddings: *Symbolic* and *Flow-Aware* that are strongly based on classic program flow analysis theory and evaluate them on two compiler optimizations tasks: Heterogeneous device mapping and Thread coarsening.
- Our novel methodology of encodings is *highly scalable* and performs better than the state-of-the-art techniques. We achieve an improved training time (up to 8000× reduction), our method is *non-data-hungry*, and it *does not encounter* Out-Of-Vocabulary (OOV) words.

This article is organized as follows: In Section 2, we discuss various related works and categorize them. In Section 3, we give the necessary background information. In Section 4, we explain the methodology for constructing the *Symbolic* and *Flow-Aware* encodings at various levels. In Section 5, we show our experimental setup followed by the discussion of results: first, we discuss the effectiveness of our seed embeddings, followed by our analysis and discussion on device mapping, and thread coarsening. In Section 6, we discuss our perspectives of IR2Vec, focusing on training time, time to generate encodings and OOV issues. Finally, in Section 7, we conclude this article.

## 2 RELATED WORKS

Modeling code as a *distributed vector* involves representing the program as a vector, whose individual dimensions cannot be distinctly labeled. Such a vector is an approximation of the original program, whose semantic meaning is "distributed" across multiple components. In this section, we categorize some of the existing works that model codes, based on their representations, the

applications that they handle, and the embedding techniques that they use. Then, we discuss the details of some specific recent works in this theme.

### 2.1 Representations, Applications and Embeddings

*Representations.* Programs are represented using standard syntactic formats like lexical tokens [2, 6, 21], Abstract Syntax Trees (ASTs) [8, 11, 51], and standard semantic formats like Program Dependence Graphs [4], and abstracted traces [30]. Then, a neural network model like RNN or its variants is trained on the representation to form distributed vectors.

We use LLVM IR [37] as the *base representation* for learning the embeddings in high-dimensional space. To the best of our knowledge, we are the *first ones* to model the entities of the IR—Opcodes, Operands and Types—in the form of relationships and to use a translation-based model [16] to capture such multi-relational data in higher dimensions.

*Applications.* In the earlier works, the training to generate embeddings was either application-specific or programming language-specific: Allamanis et al. [2] propose a token-based neural probabilistic model for suggesting meaningful method names in Java; Cummins et al. [21] propose the DeepTune framework to create a distributed vector from the tokens obtained from code to solve the optimization problems like thread coarsening and device mapping in OpenCL; Alon et al. [9] propose code2vec, a methodology to represent codes using information from the AST paths coupled with attention networks to determine the importance of a particular path to form the code vector for predicting the method names in Java; Mou et al. [42] propose a tree-based CNN model to classify C++ programs; Gupta et al. [25] propose a token-based multi-layer sequence to sequence model to fix common C program errors by students; Other applications like learning syntactic program fixes from examples [49], bug detection [46, 61] and program repair [63] model the code as an embedding in a high dimensional space followed by using RNN like models to synthesize fixes. The survey by Allamanis et al. [3] covers more such application-specific approaches.

On the other hand, our approach is more generic, and both application and programming language independent. We show the effectiveness of our embeddings on two optimization tasks (device mapping and thread coarsening) in Section 5. We believe that the scope of our work can be extended beyond these applications, including program classification, code search, prediction of vectorization/unroll factors, and the like.

*Embedding Techniques.* In encoding, entities are transformed into any numerical form amenable to learning, while in embedding, it is transformed to real-valued high-dimensional vectors. Several attempts [7, 13, 58] have been made to represent programs as distributed vectors in continuous space using word embedding techniques for diverse applications. Henkel et al. [30] use the Word2Vec embedding model to generate the representation of a program from symbolic traces. They generate and expose embeddings for the program [13, 30], or the embeddings themselves become an implicit part of the training for the specific downstream task [9, 46].

Our framework exposes a hierarchy of representations at the various levels of the program—Instruction, Function, and Program level. Our approach is the first one to propose using seed embeddings. More significantly, we are the *first ones* to use program analysis (flow-analysis)-driven approaches—*not* machine learning based approaches—to form the agglomerative vectors of programs beginning from the base seed encodings.

### 2.2 Similar Works

The closest to our work is Ben-Nun et al.'s Neural Code Comprehensions (NCC) [13], who represent programs using LLVM IR. They use skip-gram model [43] on conteXtual Flow Graph (XFG), which models the data/control flow of the program to represent IR. The skip-gram model is trained

to generate embeddings for every IR instruction (inst2vec). So as to avoid Out Of Vocabulary (*OOV*) statements, they maintain a large vocabulary, one which uses large (>640*M*) number of XFG statement pairs. A more thorough comparison of our work with NCC [13] (along with DeepTune [21]) is given in Section 6.

The recent work by Brauckmann et al. [11] represents programs as Abstract Syntax Trees and annotates it with the control and data flow edges. A Graph Neural Network is used to learn the representation of the program in a supervised manner to solve the specific tasks of device mapping and thread coarsening. We achieve better performance on both the tasks, whereas they fail to show improvements in the prediction of the thread coarsening factor.

Another recent work is ProGraML [17], which constructs a flow graph with data and control flow information from the Intermediate Representation of the program. They use inst2vec embeddings [13] to represent the nodes of the graph, and use a Gated Graph Neural Network to represent the program.

## 3 BACKGROUND

### 3.1 LLVM and Program Semantics

LLVM is a compiler infrastructure that translates source-code to machine code by performing various optimizations on its Intermediate Representation (LLVM IR) [35]. LLVM IR is a typed, well-formed, low-level, *Universal IR* to represent any high-level language and translate it to a wide spectrum of targets [37]. Being a successful compiler, LLVM provides easy access to existing control and data flow analysis and lets new passes (whether analyses or transformations) to be added seamlessly.

The building blocks of LLVM IR include Instruction, Basic block, Function, and Module. Every instruction contains opcode, type, and operands, and each instruction is statically typed. A basic block is a maximal sequence of LLVM instructions without any jumps. A collection of basic blocks form a function, and a module is a collection of functions. This hierarchical nature of LLVM IR representation helps in obtaining embeddings at the corresponding levels of the program. Characterizing the flow of information which flows into (and out of) each basic block constitutes the data flow analysis. We study the impact of using such flow information as a part of the encodings.

### 3.2 Representation Learning

The effectiveness of a machine learning algorithm depends on the choice of data representation and on the specific features used. As discussed earlier, Representation Learning is a branch of machine learning that learns the representations of data by automatically extracting the useful features [10]. Unsupervised models for learning distributed representations broadly fall under two major categories:

(1) *Context-Window-Based Embedding.* Methods such as Word2Vec [39], GloVe [47] fall under this category.
(2) *Knowledge Graph Embedding.* Methods such as TransE [16], TransR [36], TransD [32], TransH [62] fall under this category.

The context-window-based models operate on the basis of the surrounding context. The Knowledge graph embedding methods guide the learning of the entity representations based on the relationships that they participate in.

For program representations, we feel that it is important to consider semantic relationships rather than considering surrounding contexts; the latter could just mean the neighboring instructions. So, we prefer knowledge graph embedding models over context window embedding

approaches. These knowledge graph embedding models use relationships to group similar datapoints together.

*Knowledge Graph Representations.* A Knowledge Graph (KG) is a collection of (a) entities and (b) relationships between pairs of entities. Let $\langle h, r, t\rangle$ be a triplet from the knowledge graph, where the entities $h$ and $t$ are connected by relationship $r$, and their representations are learned as *translations* from head entity $h$ to the tail entity $t$ using the relation $r$ in a high-dimensional embedding space. For the same reason, these models are termed *translational*.

The representations are nothing but the vectors of pre-defined dimensions that are determined automatically by a learning method. The learning method is based on the principle that: given (representations of) any two items from the triplet (like $\langle h, r\rangle$, $\langle r, t\rangle$, $\langle t, h\rangle$), it should be possible to compute/predict the third item. Based on the above principle, different representation learning algorithms for knowledge graphs model the relationships between $h$, $r$, and $t$ in the triplets in different ways.

*Using TransE.* Of the many varieties available for KG embeddings, we use TransE [16], a *translational representation learning* model, which embeds $h$, $r$ and $t$ on to the same high dimensional space. It tries to learn the representations using the relationships of the form $h + r \approx t$, for a triplet $\langle h, r, t\rangle$.

This is achieved by following a *margin-based ranking* loss $\mathcal{L}$, where the distance between $(h + r, t)$ and $(h' + r, t')$ (where, $\langle h', r, t'\rangle$ are invalid triplets) is at least separated by a margin $m$. It is given as:

$$\mathcal{L} = \sum_{\langle h,r,t\rangle} \sum_{\langle h',r,t'\rangle} [m + distance(h + r, t) - distance(h' + r, t')]_+$$

Here, $[x]_+$ denotes the hinge loss.

Among the other Knowledge Graph models, TransE has relatively much smaller number of parameters to be learned, and has been used successfully in various algorithms for learning representations. Adopting this for our setting ensures that the representations can be learned in a faster and effective manner that scales well on huge datasets [28, 32].

### 3.3 Necessity for a LLVM-Based Embedding for Optimizations

Traditionally, machine independent compiler optimizations were always considered to be part of the "middle-end"[44] of compilers. The most successful design of this language and architecture-independent representations has been the LLVM IR [35, 37]. Using the LLVM-infrastructure, carefully well-crafted heuristics have been implemented in various optimization passes to help in achieving better performance. As the power of using ML in compilers has been recognized [22], many ML-driven approaches have also been proposed [33] as alternatives to these heuristics. However, they have primarily been feature-based, which means that integrating them into the middle-end is prone to challenges [23], both at modeling as well as engineering level.

Hence, there is a necessity for a *language and architecture agnostic embedding based compiler framework* that bridges the gap between the need for ML-based compiler optimizations, and their adoptability in compiler infrastructures. Our work tries to bridge this particular gap between embeddings from representation learning and the LLVM compiler. We believe that our work has a unique potential to be called as a *generic compiler-based embedding*.

## 4 CODE EMBEDDINGS

In this section, we explain our methodology for obtaining code embeddings at various hierarchy levels of the IR. We first give an overview of the methodology and then describe the process of

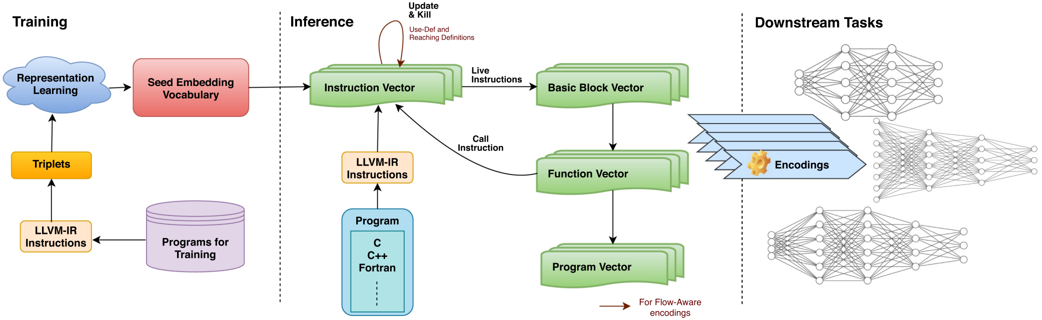



Fig. 1. Overview of IR2Vec infrastructure.

Table 1. Mapping Identifiers to Generic Representation

| Identifier | Generic representation |
|---|---|
| Variables | VAR |
| Pointers | PTR |
| Constants | CONST |
| Function names | FUNCTION |
| Address of a basic block | LABEL |

embedding instructions and basic blocks (BB) by considering the program flow information to form a cumulative *BB vector*. We then explain the process to represent the functions and modules by combining the individual BB vectors to form the final *Code Vector*. We propose two different incremental embeddings at the instruction level.

## 4.1 Overview

The overview of the proposed methodology is shown in Figure 1. Instructions in IR can be represented as an Entity-Relationship Graph, with the instruction entities as nodes, and the relation between the entities as edges. A translational learning model that we discussed in Section 3.2 is used to learn these relations (Section 4.2). The output of this learning is a dictionary containing the embeddings of the entities and is called *Seed embedding vocabulary*.

The above dictionary is looked up to form the embeddings at various levels of the input program. At the coarsest level, instruction embeddings are obtained just by using the *Seed embedding vocabulary*. We call such encodings as *Symbolic* encodings. We use the *Use-Def* and *Reaching definition* [29, 44] information to form the instruction vector for *Flow-Aware* encodings.

The instructions which are *live* are used to form the *Basic block vector*. This process of formation of a basic block vector using the flow analysis information is explained in Section 4.3. The vector to represent a function is obtained by using the basic block vectors of the function. The *Code vector* is obtained by propagating the vectors obtained at the function level with the `call graph` information, as explained in Section 4.3.2.

## 4.2 Learning Seed Embeddings of LLVM IR

*4.2.1 Generic Tuples.* The opcode, type of operation (int, float, etc.) and arguments are extracted from the LLVM IR. This extracted IR is preprocessed in the following way: first, the identifier information is abstracted out with more generic information, as shown in Table 1. Next, the Type

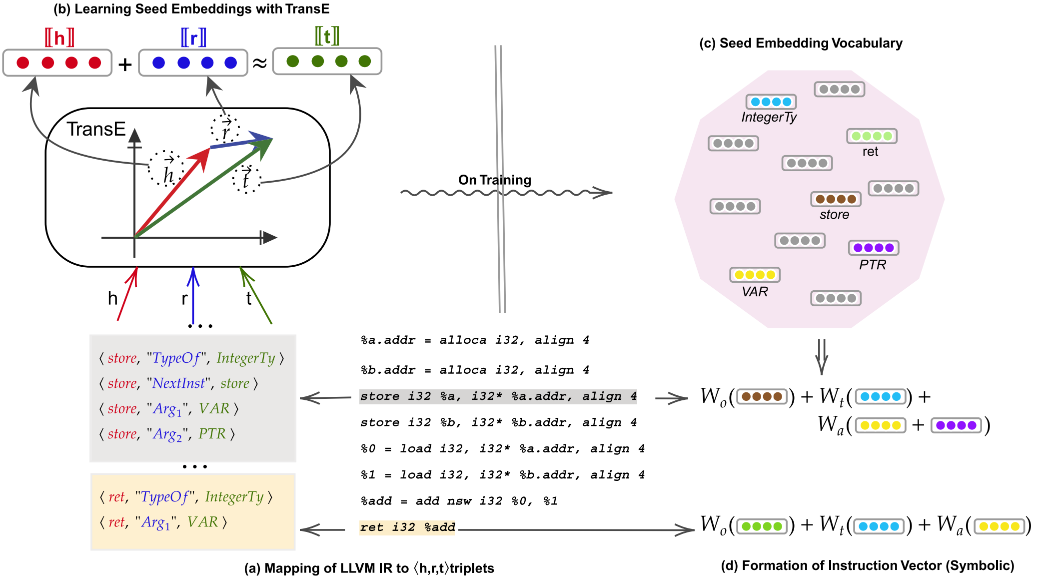



Fig. 2. Overall schematic of IR2Vec: (a) [Data Collection] Mapping LLVM IR to $\langle h, r, t\rangle$ triplets. (b) [Vocabulary training] Learning representations using TransE. (c) [After Training] Obtained Seed Embedding Vocabulary. (d) [Inference] Formation of instruction vectors for new programs.

information is abstracted to represent a base type ignoring its width. For example, the type `i32` of LLVM IR is represented as `int`.

#### 4.2.2 *Code Triplets.*

From this preprocessed data, three major relations are formed: (1) `TypeOf`: Relation between the opcode and the type of the instruction, (2) `NextInst`: Relation between the opcode of the current instruction and opcode of the next instruction; and (3) $\texttt{Arg}_i$: Relation between opcode and its $i$th operand. This transformation from actual IR to relation ($\langle h, r, t\rangle$ triplets) is shown in Figure 2(a). These triplets form the input to the representation learning model.

*Example.* The LLVM IR shown in Figure 2 corresponds to a function that sums up the two integer arguments and returns the result. The first `store` instruction of LLVM IR in Figure 2(a) is of integer type with a variable as the first operand and a pointer as the second operand. It is transformed into the corresponding triplets involving the relations `TypeOf`, `NextInst`, $\texttt{Arg}_1$, $\texttt{Arg}_2$, as shown.

#### 4.2.3 *Learning Seed Embedding Vocabulary.*

As shown in Figure 2(b), the generated Code triplets $\langle h, r, t\rangle$ are used as input to the TransE learning model (Section 3.2). On training, the model learns the representations of the entities forming the *Seed embedding vocabulary*, as shown in Figure 2(c).

### 4.3 Instruction Vector

Let the entities of instruction $l$, be represented as $O^{(l)}$, $T^{(l)}$, ${A_i}^{(l)}$ - corresponding to Opcode, Type and $i$th Argument of the instruction and their corresponding vector representations from the learned *seed embedding vocabulary* be $[\![\mathbf{O}^{(l)}]\!]$, $[\![\mathbf{T}^{(l)}]\!]$, $[\![\mathbf{A_i}^{(l)}]\!]$. Then, an instruction of format

$$\left\langle O^{(l)}\ T^{(l)} {A_1}^{(l)} {A_2}^{(l)} \cdots {A_n}^{(l)} \right\rangle$$

is represented as a vector which is computed as:

$$W_o.[\![\mathbf{O}^{(l)}]\!] + W_t.[\![\mathbf{T}^{(l)}]\!] + W_a.\left([\![\mathbf{A_1}^{(l)}]\!] + [\![\mathbf{A_2}^{(l)}]\!] + \cdots + [\![\mathbf{A_n}^{(l)}]\!]\right) \tag{1}$$

Here, $W_o$, $W_t$, and $W_a$ are scalars ($\in [0, 1]$), the plus (+) denotes the element-wise vector addition operator, and the dot (.) denotes the scalar multiplication operator. Further, the $W_o$, $W_t$, and $W_a$ are chosen with a heuristic that gives more weightage to opcode than type, and more weightage to type than arguments:

$$W_o > W_t > W_a \tag{2}$$

This resultant vector that represents an instruction is the *Instruction vector* in **Symbolic encodings**.

*Example (contd.).* For the `store` instruction shown in Figure 2(d), the representations of opcode `store`, type *IntegerTy*, and arguments *VAR*, *PTR* are fetched from the seed embedding vocabulary, and the instruction is represented as $W_o.([\![\mathbf{store}]\!]) + W_t.([\![\mathbf{IntegerTy}]\!]) + W_a.([\![\mathbf{VAR}]\!] + [\![\mathbf{PTR}]\!])$. In the same figure, we also show a similar example of the `return` instruction.

*4.3.1 Embedding Data Flow Information.* An instruction in LLVM IR may define a variable or a pointer that could be used in another section of the program. The set of uses of a variable (or pointer) gives rise to the *use-def* (UD) information of that particular variable (or pointer) in LLVM IR [29, 44]. In imperative languages, a variable can be redefined; meaning, in the flow of the program, it has a different set of lifetimes. Such a redefinition is said to *kill* its earlier definition. During the flow of program execution, only a subset of *live* definitions would reach the use of the variable. Those definitions that *reach* the use of a variable are called its *Reaching Definitions*.

We model the instruction vector using such flow analyses information to form ***Flow-Aware*** encodings. Each $A_j$, which has already been defined, is represented using the embedding of its *reaching definitions*. The Instruction Vector for a reaching definition, if not calculated, is computed in a demand-driven manner.

*4.3.2 Instruction Vector for Flow-Aware Encodings.* If $RD_1, RD_2, \ldots, RD_n$ are the reaching definitions of $A_j{}^{(l)}$, and $[\![\mathbf{RD_i}]\!]$ be their corresponding representations, then, the encoding $[\![\mathbf{A_j}^{(l)}]\!]$ is calculated by aggregating over all the vectors of the reaching definitions as follows:

$$[\![\mathbf{A_j}^{(l)}]\!] = \sum_{i=1}^{n} [\![\mathbf{RD_i}]\!] \tag{3}$$

The $\Sigma$ stands for the vector sum of the operands.

For the cases where the definition is not available (for example, function parameters), the generic entity representation of "VAR" or "PTR" from the learned *seed embedding vocabulary* is used.

An illustration is shown in Figure 4(a), where the instructions $I_{Source1_1}$ and $I_{Source2}$ reach $I_{Target}$ as arguments. Here, the definition of $I_{Source1_1}$ could reach the argument $A_{Target}$ of the instruction $I_{Target}$ either directly or via $I_{Source1_2}$. And, definition $I_{Source2}$ reaches the argument $A_{Target}$ of the instruction $I_{Target}$ directly. Hence, $[\![\mathbf{A_{Target}}]\!]$ is computed as $[\![\mathbf{I_{Source1_1}}]\!] + [\![\mathbf{I_{Source2}}]\!]$.

An instruction is said to be *killed* when the return value of that instruction is redefined. As LLVM IR is in SSA form [18, 35], each variable has a single definition, and the memory gets (re-)defined. Based on this, we categorize the instructions into two classes: ones which define memory, and ones that do not. The first class of instructions is *Write* instructions, and the second class of instructions is *Read* instructions.

For each instruction, the embeddings are formed as explained above. If these embeddings correspond to a write instruction, future uses of the redefined value will take the embedding of the current instruction, instead of the embedding corresponding to its earlier (killed) definition, until

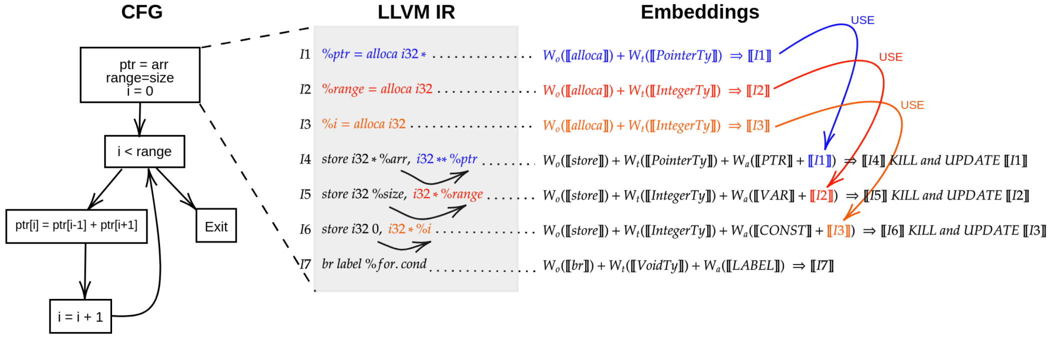
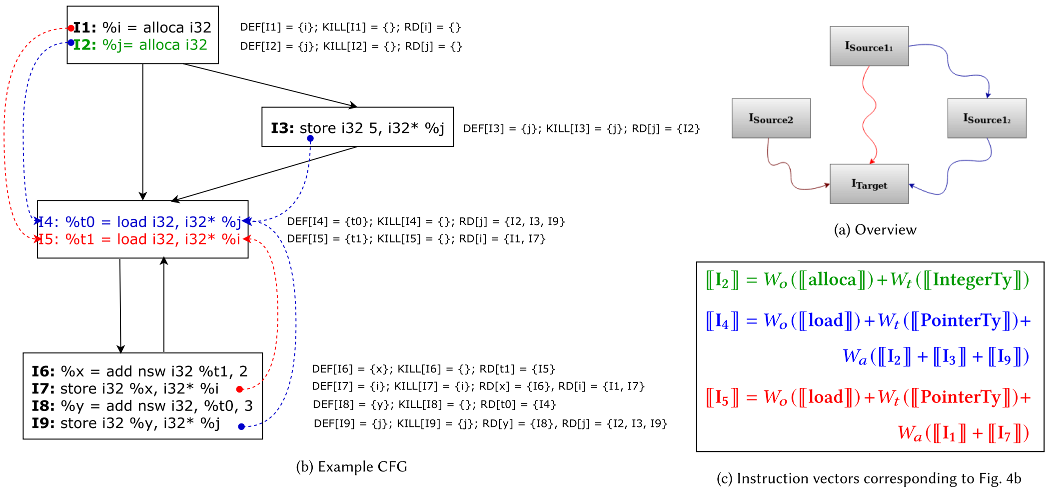



Fig. 3. Illustration of generating intra-basic block Instruction Vectors.




(a) Overview

(b) Example CFG

(c) Instruction vectors corresponding to Fig. 4b

Fig. 4. Illustration of generating inter-basic block Instruction Vectors.

the current definition itself gets redefined. This process of *Kill and Update*, along with the *use* of *reaching definitions* for forming the instruction vectors within (and across) the basic block is illustrated in Figure 3 (and Figure 4) for the corresponding Control Flow Graph (CFG), respectively.

*Example (contd.).* The CFG in Figure 3 corresponds to a function that takes `arr` and `size` as its two arguments. We expand the first basic block of this CFG and show its corresponding LLVM IR. The values of the two arguments are allocated memory in $I1$ and $I2$; this is followed by storing them to the local variables, `ptr` and `range` by the `store` instructions in $I4$ and $I5$. Similarly, memory for the loop induction variable `i` is allotted by $I3$ and is initialized to zero by $I6$.

Here, we show the process of propagating instruction vectors within the basic block.

*Example (contd.).* The definition of `ptr` in $I1$ reaches the use of `ptr` in $I4$. So, the representation of $I1$ is used in its place as shown, instead of the generic representation of PTR. Also, the store instruction kills the definition of `ptr` in $I1$, as it updates the value of `ptr` with that of `arr`. Hence, in the further uses of `ptr`, the representation of $I4$ is used instead of $I1$. The same is the case for other store instructions in $I5$ and $I6$.

Similarly, in Figure 4, we show how the propagation of instruction vectors happens across basic blocks. In this case, more than one definition can reach a use via multiple control paths. Hence, as

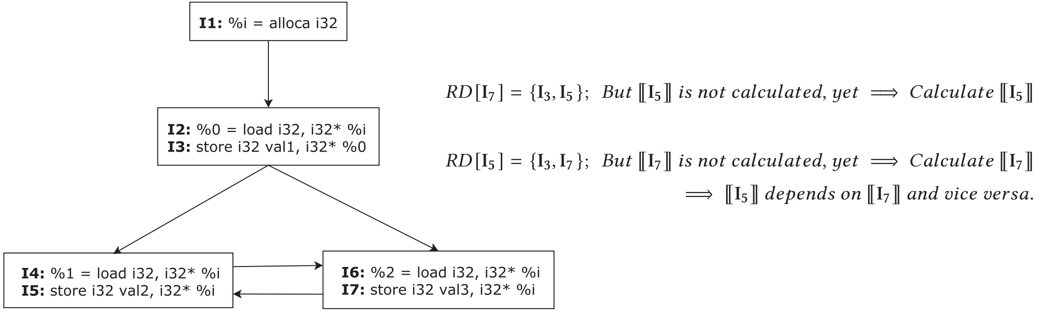



Fig. 5. Control Flow Graph showing circular dependency.

explained in Equation (3), all possible definitions that can potentially reach a use are considered to represent the argument of that instruction.

*Example (contd.).* In Figure 4(b), the definitions of `j` from $I2$, $I3$, and $I9$ can reach argument `j` of instruction $I4$ without being killed. Hence, we conservatively consider all three definitions for representing the argument `j` in $I4$ as shown in Figure 4(c). Figure 4(b) also shows another such example for instruction $I5$.

This resulting instruction vector, which is formed by adding the exact flow information on the *Symbolic* encodings' instruction vector, is used in obtaining ***Flow-Aware*** encodings.

*4.3.3 Resolving Circular Dependencies.* While forming the instruction vectors, circular dependencies between two write instructions may arise if both of them write to the same location and the (re-)definitions are reachable from each other.

*Example.* To calculate $[\![\mathbf{I_4}]\!]$ (the encoding of $I_4$), in the CFG shown in Figure 5,[1] it can be seen that the definitions of `i` from $I3$ and $I7$ can reach the argument `i` in $I4$. However, $[\![\mathbf{I_5}]\!]$ is needed for computing $[\![\mathbf{I_7}]\!]$ and is yet to be computed. But for computing $[\![\mathbf{I_5}]\!]$, $[\![\mathbf{I_7}]\!]$ is needed. Hence, this scenario results in a circular dependency.

This problem can be solved by posing the corresponding embedding equations as a set of simultaneous equations to a solver. For example, the embedding equations of $I_5$ and $I_7$ shown in Figure 5 would be:

$$\begin{aligned}
[\![\mathbf{I_4}]\!] &= W_o([\![\mathbf{load}]\!]) + W_t([\![\mathbf{IntegerTy}]\!]) + W_a([\![\mathbf{I_3}]\!] + [\![\mathbf{I_7}]\!]) \\
[\![\mathbf{I_5}]\!] &= W_o([\![\mathbf{store}]\!]) + W_t([\![\mathbf{IntegerTy}]\!]) + W_a([\![\mathbf{VAR}]\!]) + W_a([\![\mathbf{I_3}]\!] + [\![\mathbf{I_7}]\!]) \\
[\![\mathbf{I_7}]\!] &= W_o([\![\mathbf{store}]\!]) + W_t([\![\mathbf{IntegerTy}]\!]) + W_a([\![\mathbf{VAR}]\!]) + W_a([\![\mathbf{I_3}]\!] + [\![\mathbf{I_5}]\!])
\end{aligned} \tag{4}$$

It can be seen that Equation (4) is a system of linear equations, where $[\![\mathbf{I_4}]\!]$, $[\![\mathbf{I_5}]\!]$, and $[\![\mathbf{I_7}]\!]$ are the unknowns that are to be calculated, while the rest of the values are known. Such embedding equations form a system of linear equations that can be posed to a solver to find the solution.

Just like any system of linear equations, there are three cases of solutions.

(1) *Unique Solution*: In this case, the solution is obtained in a straightforward manner.
(2) *Infinitely Many Solutions*: In this case, any one of the obtained solutions can be considered as the result.
(3) *No Solution*: In this case, to obtain a solution, we perturb the value of $W_a$ as $W_a = W_a - \delta$ so that the modified system converges to a solution, with $\delta$ chosen randomly. If the system

[1]Though this CFG is the classic irreducibility pattern [29, 44], it is easy to construct reducible CFGs which have circular dependencies.

**Procedure** getInstrVec(Instruction I, Dictionary seedEmbeddings)

```
if ⟦I⟧ is already computed then
    return ⟦I⟧
⟦O⟧ ← seedEmbeddings[Opcode(I)]            // Fetch value of Opcode of I from seed embeddings
⟦T⟧ ← seedEmbeddings[Type(I)]              // Fetch value of Type of I from seed embeddings
⟦A⟧ ← ∅                                    // Initializing n-d Argument vector to zeroes
for each argument A_i ∈ I do
    if A_i is a Function then
        ⟦A_i⟧ ← seedEmbeddings["FUNCTION"]
        if Encoding is Flow-Aware and definition of A_i is available then
            ⟦A_i⟧ ← getFuncVec(A_i)
    else if A_i ∈ {VAR, PTR} then
        if Encoding is Symbolic or A_i is not a use of a definition then
            ⟦A_i⟧ ← seedEmbeddings["VAR" or "PTR"]
        else
            for each reaching definition RD of A_i do
                ⟦RD⟧ ← getInstrVec(RD)
                if ⟦RD⟧ leads to cyclic dependency then
                    Resolve and obtain ⟦RD⟧ as shown in 4.3.3
                ⟦A_i⟧ ← ⟦A_i⟧ + ⟦RD⟧
    else if A_i is a CONST then
        ⟦A_i⟧ ← seedEmbeddings["CONST"]
    else if A_i is a address of Basic block then
        ⟦A_i⟧ ← seedEmbeddings["LABEL"]
    ⟦A⟧ ← ⟦A⟧ + ⟦A_i⟧
return W_o ∗ ⟦O⟧ + W_t ∗ ⟦T⟧ + W_a ∗ ⟦A⟧
```

**Procedure** getFuncVec(Function F, Dictionary seedEmbeddings)

```
if ⟦F⟧ is already computed then
    return ⟦F⟧
⟦F⟧ ← ∅                                    // Initializing n-d Function vector to zeroes
for each basic block BB_i ∈ F do
    ⟦BB_i⟧ ← ∅                             // Initializing n-d Basic block vector to zeroes
    for each live instruction I ∈ BB_i do
        ⟦I⟧ ← getInstrVec(I, seedEmbeddings)
        ⟦BB_i⟧ ← ⟦BB_i⟧ + ⟦I⟧
    ⟦F⟧ ← ⟦F⟧ + ⟦BB_i⟧
return ⟦F⟧
```

does not converge with the chosen value of $\delta$, another $\delta$ could be picked, and the process can be iterated until the system converges.

In our entire experimentation setup described in Section 5 however, we did not encounter cases (2) and (3).

### 4.4 Construction of Code Vectors from Instruction Vector

After computing the instruction vector for every instruction of the basic block, we compute the cumulative Basic Block vector by using the embeddings of those instructions that are not killed.

For a basic block $BB_i$, the representation is computed as the sum of the representations of *live* instructions $LI_1, LI_2, \ldots, LI_m$ in $BB_i$.

$$[\![\mathbf{BB_i}]\!] = \sum_{k=1}^{m} [\![\mathbf{LI_k}]\!] \tag{5}$$

The vector to represent a function $F$ with basic blocks $BB_1, BB_2, \ldots, BB_b$ is calculated as the sum of vectors of all its basic blocks as:

$$[\![\mathbf{F}]\!] = \sum_{i=1}^{b} [\![\mathbf{BB_i}]\!]. \tag{6}$$

Our encoding and propagation also take care of programs with function calls; the embeddings are obtained by using the call graph information. For every function call, the function vector for the callee function is calculated, and this value is used to represent the call instruction. For the functions that can be resolved only during link time, we just use the embeddings obtained for the `call` instruction. The final vector that is obtained encodes the function. The above procedure is applicable for recursive function calls as well. This process of obtaining the instruction vector and function vector is outlined in Algorithms 1 and 2.

The procedure `getInstrVec` (Algorithm 1) computes the vector representation for a particular instruction $I$. The representations of the opcode ($[\![\mathbf{O}]\!]$) and type ($[\![\mathbf{T}]\!]$) of the instruction are fetched from the seed embedding vocabulary. For computing the representation of $[\![\mathbf{A}]\!]$, we iterate over the list of arguments of the instruction. If the argument corresponds to a function and the definition of that function is available, we find the representation of that function and use it as the representation of the argument. If the definition of the function is not available, then the generic representation of function in the seed embedding vocabulary is used to represent the argument.

If the argument is a variable or pointer, we compute the representation of all its reaching definitions and sum them up to represent the argument, in case of flow aware encodings. For symbolic encodings, we use a simpler procedure by using the generic representation of variable or pointer from the seed embedding vocabulary as the representation of the argument.

For the other cases, when the argument is a constant or an address of a basic block (its label), the corresponding generic representations are used. Finally, representations of all the arguments (of the instruction) are summed up to compute $[\![\mathbf{A}]\!]$. And, the instruction vector is computed as the weighted sum of $[\![\mathbf{O}]\!]$, $[\![\mathbf{T}]\!]$, and $[\![\mathbf{A}]\!]$.

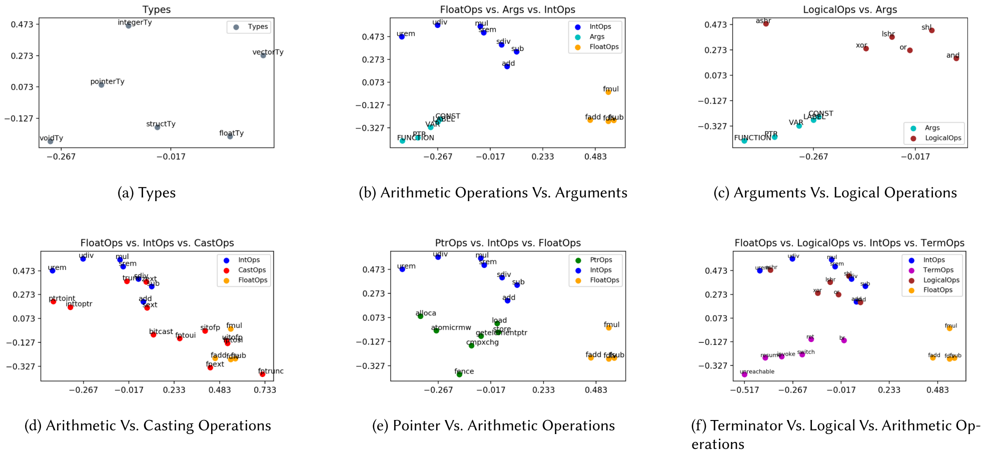
The procedure `getFuncVec` (Algorithm 2) computes the function vector. First, the representation of every basic block in the function is calculated. The instruction vectors corresponding to the live instructions are calculated by making a call to the procedure `getInstrVec` and the basic block vector is obtained as the element-wise sum of these instruction vectors, which, when summed up, forms the function vector.

If $[\![\mathbf{F_1}]\!], [\![\mathbf{F_2}]\!], \ldots, [\![\mathbf{F_f}]\!]$ are the embeddings of the functions $F_1, F_2, \ldots, F_f$ in a program, then the code vector representing the program $P$ is calculated as the sum of the embeddings of all such functions as:

$$[\![\mathbf{P}]\!] = \sum_{i=1}^{f} [\![\mathbf{F_i}]\!] \tag{7}$$

## 5 EXPERIMENTAL RESULTS

We used the SPEC CPU 17 [12] benchmarks and Boost library [15] as the datasets for learning the representations. The programs in these benchmarks are converted to LLVM IR by varying the compiler optimization levels (`-O[0-3], -Os, -Oz`) at random. The $\langle h, r, t\rangle$ triplets are created from the resultant IRs using the relations that were explained in Section 4.2. Embeddings of such




(a) Types (b) Arithmetic Operations Vs. Arguments (c) Arguments Vs. Logical Operations

(d) Arithmetic Vs. Casting Operations (e) Pointer Vs. Arithmetic Operations (f) Terminator Vs. Logical Vs. Arithmetic Operations

Fig. 6. Comparison of embeddings of various seed entities.

triplets are learned using an open-source implementation of TransE [28], which was explained in Section 3.2.

We wrote an LLVM analysis pass to extract the generic tuples from the IR so that they can be mapped to form triplets. There were $\approx 134M$ triplets in the dataset, out of which $\approx 8K$ relations were unique; from these, we obtain 64 different entities whose embeddings were learnt. The training was done with SGD optimizer for 1,500 epochs to obtain embedding vectors of 300 dimensions.

These learned embeddings of 64 different entities form the *seed embeddings*. Additional related information is listed in Table 5. We heuristically set $W_o$, $W_t$, and $W_a$ to 1, 0.5, 0.2, respectively. The vectors at various levels were computed using another LLVM pass.

In this section, we attempt to answer **RQ1** by showing the clusters formed by the entities of *seed embedding vocabulary* and **RQ2** by showing the effectiveness of IR2Vec embeddings on two different optimization tasks.

## 5.1 RQ1: Evaluation of Seed Embeddings

The seed embeddings are analyzed to demonstrate whether the semantic relationship between the entities are effectively captured by forming clusters. These clusters show that IR2Vec is able to capture the characteristics of the instructions and the arguments.

The entities corresponding to the obtained seed embeddings are categorized as groups based on the nature of the operations they perform—*Arithmetic operators* containing the integer- and floating-point-type arithmetic operations, *Pointer operators* which access memory, *Logical operators* which perform logical operations, *Terminator operators* which form the last instruction of the basic block, *Casting operators* that perform typecasting, *Type* information, and *Arguments*.

Clusters showing these groups are plotted using PCA (Principal Component Analysis), a dimensionality reduction mechanism to project the points from the higher- to lower-dimensional space [60]. Here, to visualize the 300-dimensional data in 2 dimensions, we use 2-PCA, and the resulting clusters are shown in Figure 6.

In Figure 6(a), we show the relation between various types. It can be observed that `vectorTy` being an aggregate type can accept any of the other first-class types and lies approximately equidistant from `integerTy`, `pointerTy`, `structTy`, and `floatTy`. And, `vectorTy` lies farthest

from `voidTy`, justifying that it is unlikely that `voidTy` elements will be aggregated together as vectors.

In Figure 6(b), we show that all integer-based arithmetic operators are grouped together and are distinctly separated from floating-point-based operators. It can be said that the analogies between the operators are captured. For example, the distance between (`add`, `fadd`) is similar to that of the distance between (`sub`, `fsub`). From Figures 6(b), 6(e), and 6(f), it can also be seen that the arithmetic operators are distinctly separated from the arguments, pointer operators and terminator operators. Similarly, from Figure 6(c), we can see that the logical operators are also distinctly separated from the arguments.

In Figure 6(d), we show the relationship between arithmetic and casting operators. It can be clearly seen that the integer-based casting operators like `trunc`, `zext`, `sext`, and the like, are grouped together with integer operators, and the floating-point-based casting operators like `fptrunc`, `fpext`, `fptoui`, and the like are grouped together with floating-point operators. On observing Figure 6(d) and Figure 6(e), it can be seen that `ptrtoint` and `inttotpr` are closer to both integer operators and pointer operators. Figure 6(e) also demonstrates that the arithmetic operators are clearly distinct from pointer operators. Logical operators operate on integers, and hence they are grouped together with integer operators, as observed in Figure 6(f).

In summary, these clusters show that the obtained seed embeddings are indeed meaningful as they capture intrinsic syntactic and semantic relationships of LLVM entities.

### 5.2 RQ2 (a): Heterogeneous Device Mapping

Grewe et al. [26] proposed the heterogeneous device mapping task to map OpenCL kernels to the optimal target device—CPU or GPU in a heterogeneous system. In this experiment, we use the embeddings obtained by IR2Vec to map OpenCL kernels to its optimal target.

*Dataset.* We use the dataset provided by Ben-Nun et al. [13] for this experiment. It consists of 256 unique OpenCL kernels from seven different benchmark suites comprising of AMD SDK, NPB, NVIDIA SDK, Parboil, Polybench, Rodinia and SHOC. Taking the kernels and varying their two auxiliary inputs (data size and workgroup size) for each kernel, a dataset is obtained. This leads to about 670 CPU- or GPU-labeled data points for each of the two GPU devices—AMD Tahiti 7970 and NVIDIA 970.

*Experimental Setup.* The embeddings for each kernel are computed using IR2Vec infrastructure. Gradient boosting classifier with a learning rate of 0.5, which allows a maximum depth of up to 10 levels and 70 estimators with ten-fold cross-validation is used to train the model.

We use simpler models like gradient boosting classifier, as our embeddings are generated at the program/function level directly without forming sequential data that need sequential neural networks like RNNs or LSTMs. More advantages of our modelling are discussed in Section 6.1. Similar to the earlier methods, we use the runtimes corresponding to the predicted device, to calculate the speedup against the static mapping heuristics proposed by Grewe et al. [26].

We compare the prediction accuracy and speedup obtained by IR2Vec across two platforms (AMD Tahiti 7970 and NVIDIA GTX 970) with the manual feature-extraction approach of Grewe et al. [26] and the state-of-the-art methodologies of DeepTune [21], and inst2vec [13].

*Accuracy.* Accuracy is computed as the percentage of correct device mapping predictions for a kernel by the model over the total number of predictions during test time. In Figure 7, we show the accuracy of mapping using the encodings generated by IR2Vec and other methods. *Flow-Aware* and *Symbolic* encodings achieve an average accuracy of 91.26% and 88.72% accuracy, respectively.

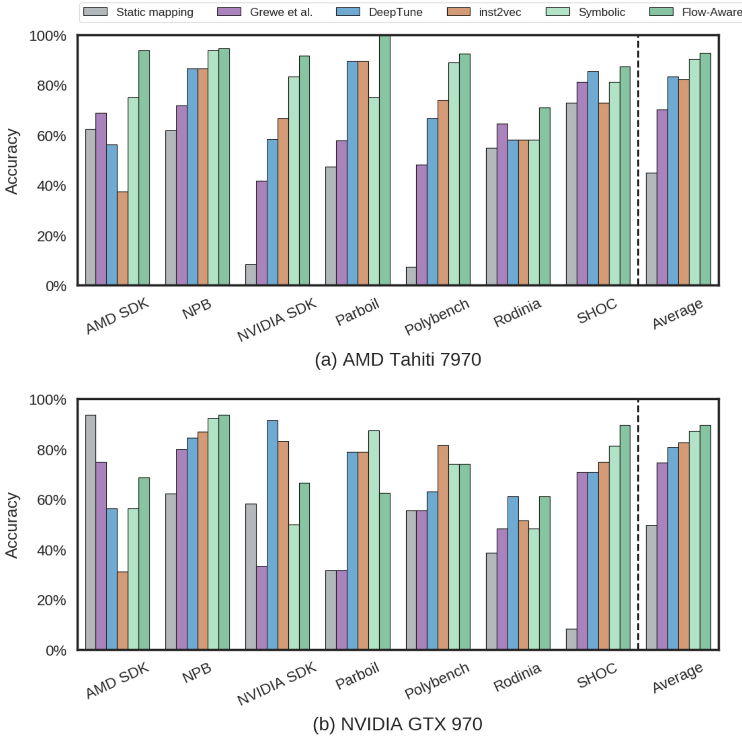



Fig. 7. Accuracy for heterogeneous device mapping task on various benchmark suites.

Table 2. Percentage of Improvement in Accuracy Obtained by Flow-Aware Encodings when Compared to the Other Methods

| **Architecture** | **Grewe et al. [26]** | **DeepTune [21]** | **inst2vec [13]** | **inst2vec-imm[2] [13]** | **IR2Vec Symbolic** |
|---|---|---|---|---|---|
| **AMD Tahiti 7970** | 26.50% | 10.93% | 12.12% | 5.38% | 2.81% |
| **NVIDIA GTX 970** | 22.96% | 11.70% | 9.69% | 3.54% | 2.92% |

In Table 2, we show the percentage improvement of accuracy obtained by *Flow-Aware* encodings over the other methods. It can be observed that the *Flow-Aware* encodings achieve higher performance over the other methods [13, 21, 26]. On AMD Tahiti 7970, our *Flow-Aware* encodings achieve the highest accuracy in all 7 benchmark suites. In addition, our *Symbolic* encodings achieve second-highest in 4/7 benchmark-suites. Similarly, in NVIDIA GTX 970, *Flow-Aware* encodings achieve the highest accuracy in 3/7 cases, and *Symbolic* encodings achieve the highest or second-highest accuracy in 4/7 cases.

Along with inst2vec encodings, the NCC framework [13] also proposes the inst2vec-imm encodings to handle immediate values. For this, they formulate four *immediate value handling methods*. As there is no single consistent winner among these value handling methods, NCC considers the best result out of them. As the benchmark-wise results are not published, we are unable to compare our results with inst2vec-imm in detail.

[2]The numbers are quoted from the paper.

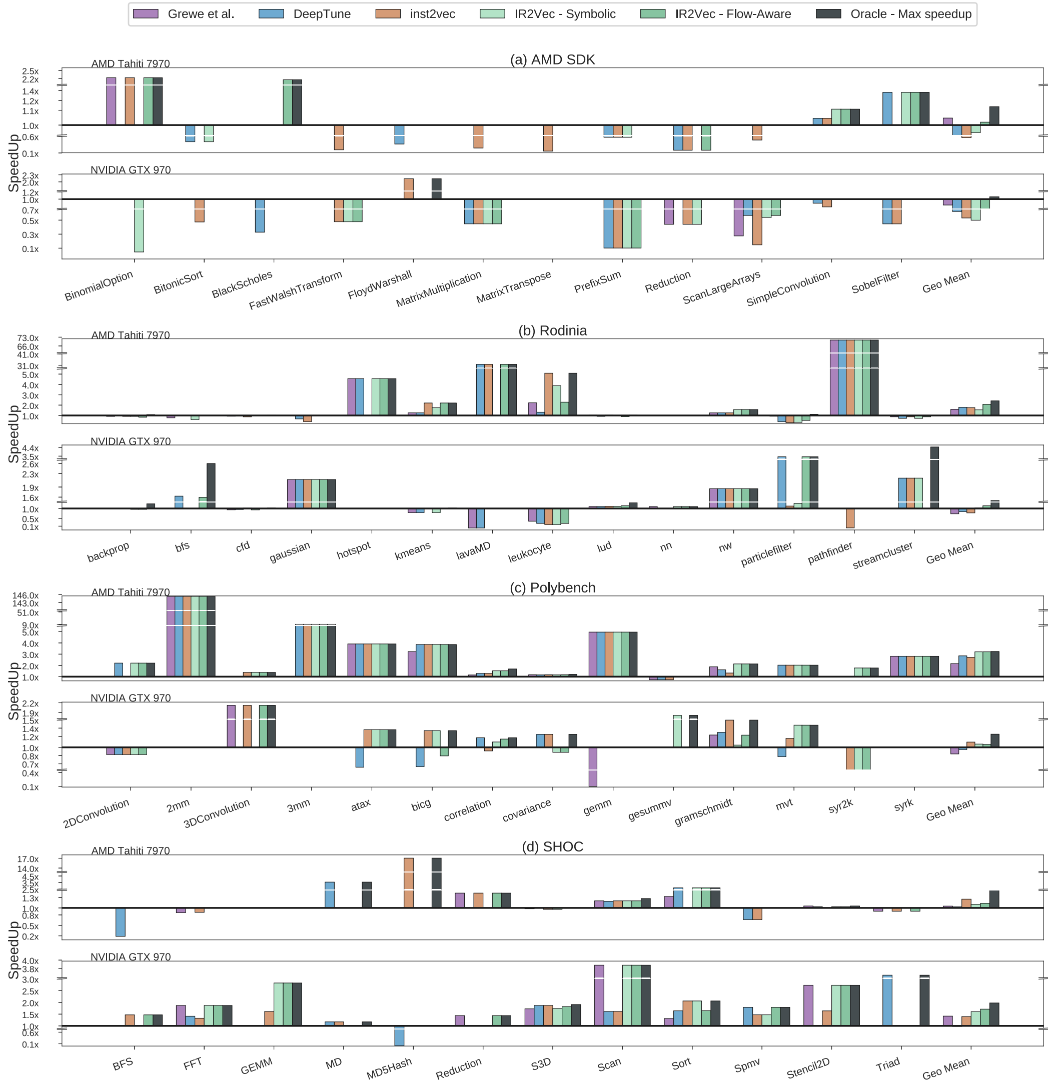



Fig. 8. Plot showing the speedups corresponding to the device mapping predictions by various methods.

*Speedups.* In Figure 8, we show the speedups achieved in comparison with static mapping as the baseline on both the platforms under consideration. With *Flow-Aware* encodings, we achieve about 88.38% and 77.65% of the maximum speedup given by the oracle on AMD Tahiti and NVIDIA GTX resepectively, when compared to the 70.76% on AMD Tahiti, 70.66% on NVIDIA GTX by inst2vec, 72.90% on AMD Tahiti, and 66.79% on NVIDIA GTX by Deeptune. (With *Symbolic* encodings, we achieve about 80.11% and 72.03% of the maximum speedup on AMD Tahiti and NVIDIA GTX.)

*Flow-Aware* encodings achieve a geometric mean of 1.58× speedup on AMD Tahiti 7970 and a 1.26× speedup on NVIDIA GTX 970; in comparison, on the AMD platform, the speedups achieved by the state-of-the-art models of DeepTune [21] and inst2vec [13] is 1.46× and 1.4×, respectively; while, on the NVIDIA platform, both DeepTune and inst2vec achieve a comparable 1.21× speedups.

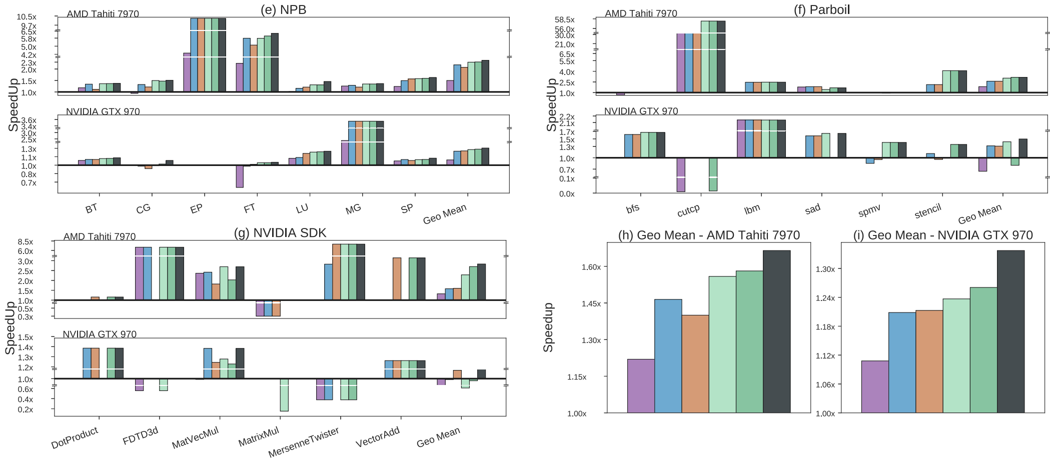



Fig. 8. *(Contd.)* Plot showing the speedups corresponding to the device mapping predictions by various methods.

Table 3. Percentage of Improvement in Speedup Obtained by Flow-Aware Encodings when Compared to the Other Methods

| Architecture | Grewe et al. [26] | DeepTune [21] | inst2vec [13][3] | IR2Vec Symbolic |
|---|---|---|---|---|
| **AMD Tahiti 7970** | 29.65% | 7.96% | 12.95% | 1.43% |
| **NVIDIA GTX 970** | 13.77% | 4.31% | 3.93% | 1.92% |

*Symbolic* encoding also performs better than the state-of-the-art by achieving a speedup of 1.56× on AMD Tahiti and 1.24× on NVIDIA GTX.

In Table 3, we show the percentage improvement of speedups obtained by *Flow-Aware* encodings in comparison with the earlier methods. It can be seen that the *Flow-Aware* encodings perform better on both the platforms when compared to the other works.

On benchmarks like ep (*embarrassingly parallel*), where mapping them to GPUs gets the optimal performance [26], we map to GPU—in all the turns of 10 fold cross-validation—and hence achieve the highest possible speedup. Similarly, for LU, which gives an optimal performance when mapped to CPUs [26], we map the kernels that ought to be mapped to CPU correctly, with very high confidence (of about 95%) in various turns of 10 fold cross-validation. In comparison, DeepTune and inst2vec map LU to CPUs with a confidence of 77% and 85%, respectively.

*Slowdown.* Our predictions using *Flow-Aware* encodings result in the *least* number of slowdowns both at the level of benchmarks and benchmark-suites. Our predictions result in a slowdown in 18/142 of benchmarks across two platforms; in comparison, inst2vec and DeepTune result in a slowdown in 33 and 32 benchmarks, respectively. Also, at an aggregate benchmark-suite level, the prediction using *Flow-Aware* encodings leads to a slowdown in 3/14 cases across platforms, in comparison inst2vec and DeepTune result in slowdowns in three and six cases, respectively.

[3]The inst2vec-imm results given in their paper [13] are based on arithmetic mean, whereas our results are based on geometric mean. inst2vec-imm is reported to achieve a (arithmetic mean) speedup of 3.47× and 1.44×; whereas our *Flow-Aware* encodings achieve a speedup of 3.51× and 1.47× on AMD Tahiti and on NVIDIA GTX respectively.

Fig. 9. Plot showing the speedups achieved by predicted coarsening factors by various methods.

## 5.3 RQ2 (b): Prediction of Optimal Thread Coarsening Factor

Thread coarsening [57] is the process of increasing the work done by a single thread by fusing two or more concurrent threads. Thread coarsening factor corresponds to the number of threads that can be fused together. Selection of an optimal thread coarsening factor can lead to significant improvements [40] in the speedups on GPU devices and a naive coarsening would lead to a substantial slowdown.

A thread coarsening factor of a kernel that gives the best speedup on a GPU could give the worst performance with the same coarsening factor on another device (either within or across vendors) because of the architectural characteristics of the device [41, 55]. For example, nbody kernel, which has a higher degree of Instruction Level Parallelism, can be better exploited by VLIW-based AMD Radeon than SIMD-based AMD Tahiti [41].

*Dataset.* In this experiment, we follow the experimental setup proposed by Magni et al. [41] to predict the optimal thread coarsening factor—among $\{1, 2, 4, 8, 16, 32\}$—for a given kernel specific to a GPU device. Even for this experiment, we use the dataset provided by Ben-Nun et al. [13]. It consists of about 68 datapoints from 17 OpenCL kernels on 4 different GPUs—AMD Radeon 5900, AMD Tahiti 7970, NVIDIA GTX 480, and NVIDIA Tesla K20c. These kernels are collectively taken from AMD SDK, NVIDIA SDK and Parboil benchmarks. A datapoint consists of the kernel and its runtime corresponding to each thread coarsening factor on a particular GPU device.

*Experimental Setup.* Even for this task, we use gradient boosting classifier instead of LSTMs and RNNs to predict the coarsening factor for the four GPU targets. For this experiment, we set the

Table 4. Percentage of Improvement in Speedup Obtained by Flow-Aware Encodings When Compared to the Other Methods

| Architecture | Magni et al. [41] | DeepTune [21] | DeepTune-TL [21] | inst2vec [13][4] | IR2Vec Symbolic |
|---|---|---|---|---|---|
| **AMD Radeon 5900** | 27.66% | 5.26% | 5.26% | 4.35% | – |
| **AMD Tahiti 7970** | 25.41% | 29.37% | 36.56% | 18.17% | 2.08% |
| **NVIDIA GTX 480** | 45.31% | 25.21% | 18.89% | 23.89% | 3.98% |
| **NVIDIA Tesla K20c** | 52.84% | 15.41% | 11.98% | 11.98% | 0.18% |

learning rate as 0.05 with 140 decision stumps with 1 level, as the number of datapoints in the dataset is very low. We use ten-fold cross-validation for measuring the performance.

*Speedups.* In Figure 9, we show the speedups achieved by our encodings and earlier works on four different platforms—AMD Radeon HD 5900, AMD Tahiti 7970, NVIDIA GTX 480, and NVIDIA Tesla K20c.

On AMD Radeon, both of our encodings achieve a speedup of 1.2× when compared to the state-of-the-art speedup of 1.15× and 1.14× achieved by inst2vec [13] and DeepTune model with transfer learning (DeepTune-TL) [21]. In AMD Tahiti, *Flow-Aware* encodings achieve a speedup of 1.23×; *Symbolic* encoding achieves a speedup of 1.2×, whereas the earlier works by DeepTune-TL and inst2vec achieve a speedup of 0.9× and 1.04× respectively. In Table 4, we show the percentage improvement of speedup obtained by *Flow-Aware* encodings over other methodologies. From the table, we can infer that *Flow-Aware* gives better results for every architecture when compared to the other methods.

We are the first ones to achieve a positive speedup on NVIDIA GTX 480; our *Flow-Aware* and *Symbolic* encodings obtain a speedup of 1.18× and 1.13×, respectively, when compared to 0.95× and 0.99× speedup achieved by inst2vec and DeepTune-TL. We get a speedup of 1.13× with both of our proposed encodings on NVIDIA Tesla K20c. In contrast, DeepTune-TL and inst2vec obtain a speedup of 1.01× on this platform.

On average, it can be seen that both encodings outperform the earlier methods for prediction of the thread coarsening factor on all the four platforms under consideration.

*Slowdown.* Magni et al. [41] observe that `spmv` and `mvCoal` kernels have irregular dependences that causes a poor response to coarsening, and hence no performance improvement for them is possible. For these kernels, IR2Vec obtains the baseline speedup *without* resulting in a slowdown. In contrast, the earlier models result in negative speedups (DeepTune results in a slowdown up to 0.36× in AMD Radeon and inst2vec results in a slowdown of up to 0.63× in AMD Radeon and NVIDIA GTX). The same argument applies for `stencil` kernel (an iterative Jacobi stencil on 3D-grid), where the coarsening leads to slowdown (except in NVIDIA GTX), while IR2Vec still obtain the baseline speedup.

When compared to the other methods, we obtain the best speedup on about 70% of the kernels on all platforms. It can be observed that the *Flow-Aware* encodings *rarely lead to slowdowns*; this happens in only 8/68 cases (17 benchmark-suits, across 4 platforms), even on these eight cases, the speedup is still close—within 10%—of the baseline. Whereas, predictions by inst2vec

[4] As per the results given in the NCC paper [13], inst2vec-imm achieves a (arithmetic) mean of 1.28×, 1.18×, 1.11×, and 1× speedup on AMD Radeon, AMD Tahiti, NVIDIA GTX and NVIDIA Tesla; whereas our *Flow-Aware* encodings achieve a (arithmetic) mean of 1.25×, 1.3×, 1.26×, and 1.16×, respectively.

and DeepTune-TL result in a slowdown in 18 and 21 cases. We believe that this is because of the flow information associated with the obtained vectors.

## 6 RQ3: IR2Vec-PERSPECTIVES

We discuss some perspectives on IR2Vec. We answer **RQ3** by doing a scalability study.

### 6.1 Training Characteristics

By design, training with IR2Vec embeddings takes lesser training time. This is because our framework has the flexibility to model the embeddings as non-sequential data at program or function level. Whereas, other methods are limited to modeling the input programs only as sequential data, and hence are bound to using sequential models like LSTMs and RNNs. Using such models will involve training more number of parameters than non-sequential models like Gradient Boosting.

*Device Mapping*. Training for the device mapping task by IR2Vec takes ≈ 5 seconds on a P100 GPU, when compared to about 10 hours and 12 hours of training time taken by DeepTune [21] and NCC [13], respectively. This results in a *reduction of about* ≈ 7200×–8640× *in training time without a reduction in performance.*

The earlier works take much time for training because they involve training a large number of parameters: DeepTune [21] uses ≈ $77K$ parameters, while NCC [13] uses ≈ $69K$ parameters. In contrast, the IR2Vec predictions use Gradient Boosting, which is a collection of a small number of shallow decision trees. This reduction in time is primarily possible because the embeddings obtained by IR2Vec enable us effectively use the Gradient boosting algorithm instead of the compute-intensive and data-hungry neural networks (LSTM in this case) that do not fit well in the cache hierarchy.

*Thread Coarsening*. Even for the thread coarsening task, our model takes lesser time of ≈10 seconds for training when compared to ≈11 hours of training time needed by DeepTune-TL and ≈1 hour of training time needed (and 77K and 69K parameters used) by DeepTune and NCC approaches. This results in ≈360×–3960× reduction of training time, and again, achieving good speedups.

### 6.2 Symbolic vs. Flow-Aware

The seed embedding vocabulary captures intrinsic syntactic and semantic relations at the entity level of the LLVM IR (Section 5.1). Hence, we believe that the primary strength of our encodings comes from the *seed embedding vocabulary*. This directly leads to the *Symbolic* encodings that achieve better performance than the other earlier methods on average. When the *Symbolic* encodings are augmented with flow information of the program, it results in *Flow-Aware* encodings. These encodings result in much more informative representation and hence lead to better accuracy and speedup than all other methods. This is evident from the results shown in Section 5.

However, this improvement in accuracy comes with a minimal overhead. Generating the *Flow-Aware* encodings take more time than *Symbolic* encoding, as it accounts for the time taken to generate and propagate the program flow information and to resolve circular dependencies in this process. We did an analysis of time taken by both of these encodings on a sample set of straightforward programs involving the family of sorting, searching, dynamic and greedy programs obtained from an online collection of programs [24].

The comparison of time taken to generate the *Symbolic* and *Flow-Aware* encodings from the *Seed Embedding Vocabulary* on the sample set is shown in Figure 10. It can be observed that, on an average, *Flow-Aware* encodings take 1.86 times more than that of *Symbolic* encodings. Their

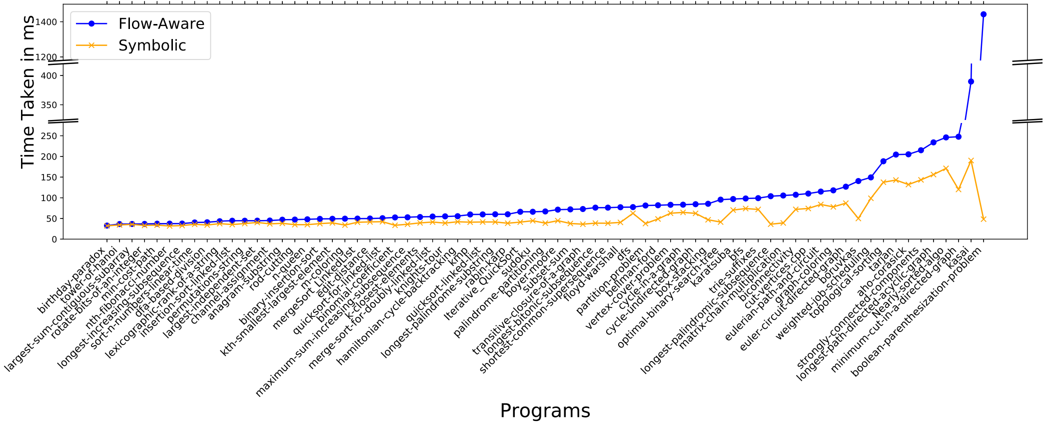



Fig. 10. Comparison of time taken to generate the Symbolic and Flow-Aware encodings from *Seed Embedding Vocabulary*.

memory representations are of the same size, as both of them result in a floating-point vector in the same n-dimensions (300 dimensions for our setting).

### 6.3 Exposure to OOV Words

For learning the representations of programs, the training phase of any method often involves learning a vocabulary that contains the embeddings corresponding to the input. When an unseen combination of underlying input constructs is encountered during inference, it would not be a part of the vocabulary and leads to *Out Of Vocabulary* (*OOV*) data points. In such cases, most of the models treat all the *OOV* words in a similar manner by assigning a common representation ambiguously, which may result in performance degradation.

Hence, to avoid *OOV* points, it is important to expose various and large (all possible) combinations of the underlying entities during the training phase. For example, for generating the embeddings at a statement-level of IR, all combinations of opcodes, types and arguments that can potentially form a statement should be exposed during training. Similarly, for generating token-based embeddings, all possible tokens that can possibly be encountered must have been exposed at the training time. But, both of these approaches would lead to a huge training space of $O(|opcodes| \times |types| \times |arguments|)$ and $O(|tokens|)$ (where $|tokens|$ can potentially be unbounded, with tokens being used more in the sense of a lexeme [5]), respectively. As can be seen, covering such a huge intractable space is infeasible, and hence undesirable.

Consequently, the methods that learn statement-level representations like NCC [13] face *OOV* issues. However, DeepTune [21] uses a Token and Character based hybrid approach to overcome this issue. DeepTune's method involves usage of the embeddings corresponding to the token if it is present in vocabulary. Else, they break the token as a series of characters and use the corresponding embeddings of the character.

On the other hand, IR2Vec forms embeddings at the entity level of the IR, and hence it is sufficient to expose a training space of only $O(|opcodes| + |types| + |arguments|)$ to avoid *OOV* points. With this insight, it can be seen that IR2Vec can avoid the *OOV* issue even on exposure to a smaller number of programs at training time, when compared to the other approaches. Consequently, this results in a smaller vocabulary, and hence achieving better performance than the other methods. A comparison of IR2Vec with DeepTune and NCC with respect to training and vocabulary is shown in Table 5.

Table 5. Comparison Matrix: DeepTune vs. NCC vs. IR2Vec

| Comparison metric | DeepTune [21] | NCC [13] | IR2Vec |
|---|---|---|---|
| **Primary embedding** | Token and Character level | Instruction level | Entity level |
| **Files examined** | Handpicked vocabulary | 24,030 | 13,029 |
| **Vocabulary size** | 128 symbols | 8,565 statement embeddings | 64 entity embeddings |
| **Entities examined** | Application specific | ≈640M XFG Statement Pairs | ≈134M Triplets |
| **Vocabulary training time** | Task dependent | 200 hrs on a P100 | 20 mins on a P100 |

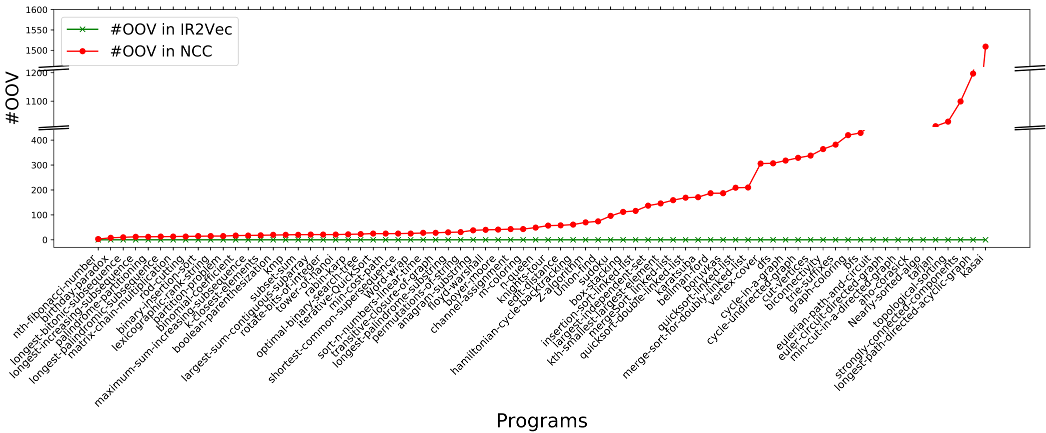



Fig. 11. Comparison of the number of *OOV* entities encountered by NCC and IR2Vec.

A comparison of the number of *OOV* entities encountered by NCC and IR2Vec on the same set of programs used in Section 6.2 is shown in Figure 11. It can be seen that our method does not encounter any *OOVs* even when exposed to lesser training data, thereby achieving good scalability.

## 7 CONCLUSIONS AND FUTURE WORK

We proposed IR2Vec, a novel LLVM-IR based framework that can capture the implicit characteristics of the input programs in a task-independent manner. The seed embeddings were formed by modelling IR as relations, and the encoding was obtained by using a translational model. This encoding was combined with liveness, use-def, and reaching definition information, to form vectors at various levels of the program abstraction like instruction, function and module. Overall, this results in two encodings, which we term as *Symbolic* and *Flow-Aware*.

When compared to earlier approaches, our approach of representing programs is *non-data-hungry*, takes *less training* time of up to 8640×, while maintaining a *small vocabulary* of *only 64 entities*. As we use entity level seed embeddings, we *do not* encounter any *OOV* issues. We demonstrate the effectiveness of the obtained encodings on two different tasks and obtain *superior* performance results while achieving *high* scalability when compared with various similar approaches.

We envision that our framework can be applied to other applications beyond the scope of this work. IR2Vec can be extended to classify whether a program is malicious or not by looking for suspicious and obfuscated patterns. It can also be applied for detecting codes with vulnerabilities, and to identify the patterns of code and replace them with its optimized equivalent library calls. It can even be extended to aide in key optimizations like the prediction of vectorization,

interleaving, and unrolling factors. We also plan to extend the device mapping and thread coarsening experiments with more datasets and on newer platforms.

The source code and other relevant material are available in http://www.compilers.cse.iith.ac.in/research/ir2vec.

## ACKNOWLEDGMENTS

We are grateful to Suresh Purini, Dibyendu Das, Govindarajan Ramaswamy and Albert Cohen, for their valuable feedback on our work at various stages. We also thank Swapnil Dewalkar, Akash Banerjee and Rahul Utkoor for the thoughtful discussions in the early stages of this work. We would like to thank the anonymous reviewers of ACM TACO for their insightful and detailed comments which helped in improving the article.

## REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, 281–293.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 38–49.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=BJOFETxR-

[5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* ($2^{nd}$ Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[6] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, (ICML 2016)*. 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html.

[7] Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 37. PMLR, Lille, France, 2123–2132. http://proceedings.mlr.press/v37/allamanis15.html.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York,, 404–419. DOI: https://doi.org/10.1145/3192366.3192412

[9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning distributed representations of code. In *Proceedings of the ACM Conference on Programming Languages (POPL)*, Vol. 3, Article 40 (Jan. 2019), 29 pages. DOI: https://doi.org/10.1145/3290353

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 8 (Aug. 2013), 1798–1828. DOI: https://doi.org/10.1109/TPAMI.2013.50

[11] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction* (Feb. 2020), 201–211. DOI: https://doi.org/10.1145/3377555.3377894

[12] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE'18)*. ACM, New York,, 41–42. DOI: https://doi.org/10.1145/3185768.3185771

[13] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., 3589–3601. http://dl.acm.org/citation.cfm?id=3327144.3327276.

[14] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. 2014. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.* 79 (Jan. 2014), 241–259. DOI: https://doi.org/10.1016/j.scico.2012.04.008

[15] Boost. 2018. Boost C++ Libraries. https://www.boost.org/. Accessed 2019-05-16.

[16] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural*

*Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., 2787–2795. http://dl.acm.org/citation.cfm?id=2999792.2999923.

[17] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. ProGraML: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536* (2020).

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[19] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York,, 964–974. DOI: https://doi.org/10.1145/3338906.3340458

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[21] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[22] Keith D. Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive optimizing compilers for the 21st century. *J. Supercomput.* 23, 1 (Aug. 2002), 7–22. DOI: https://doi.org/10.1023/A:1015729001611

[23] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. 2011. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming* 39, 3 (01 Jun 2011), 296–327. DOI: https://doi.org/10.1007/s10766-010-0161-2

[24] GeeksforGeeks. 2003. C/C++/Java programs. https://www.geeksforgeeks.org. Accessed 2019-08-15.

[25] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI'17)*. AAAI Press, 1345–1351. http://dl.acm.org/citation.cfm?id=3298239.3298436

[26] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, (CGO 2013), (Shenzhen, China, February 23-27), 2013*. IEEE Computer Society, 22:1–22:10. DOI: https://doi.org/10.1109/CGO.2013.6494993

[27] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 242–255. DOI: https://doi.org/10.1145/3368826.3377928

[28] Xu Han, Shulin Cao, Xin Lv, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. 2018. OpenKE: An open toolkit for knowledge embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Brussels, Belgium, 139–144. DOI: https://doi.org/10.18653/v1/D18-2024

[29] Matthew S. Hecht. 1977. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA.

[30] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, NY, USA, 163–174. DOI: https://doi.org/10.1145/3236024.3236085

[31] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[32] G. Ji, S. He, L. Xu, K. Liu, and J. Zhao. 2015. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 687–696. DOI: https://doi.org/10.3115/v1/P15-1067

[33] P. J. Joseph, Matthew T. Jacob, Y. N. Srikant, and Kapil Vaswani. 2007. Statistical and machine learning techniques in compiler design. In *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, Y. N. Srikant and Priti Shankar (Eds.). CRC Press.

[34] V. Kashyap, D. B. Brown, B. Liblit, D. Melski, and T. Reps. 2017. Source forager: A search engine for similar source code. *arXiv preprint arXiv:1706.02769* (2017).

[35] Chris Lattner and Vikram Adve. 2004. 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*. IEEE Computer Society, 75.

[36] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu. 2015. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI'15)*. AAAI Press, 2181–2187.

[37] LLVM. 2018. LLVM Language Reference. https://llvm.org/docs/LangRef.html. Accessed 2019-08-20.

[38] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.* 3, (OOPSLA), Article 152 (Oct. 2019), 28 pages. DOI: https://doi.org/10.1145/3360578

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[40] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13, Denver, CO, USA - November 17-21, 2013)*. William Gropp and Satoshi Matsuoka (Eds.). ACM, 11:1–11:11. DOI: https://doi.org/10.1145/2503210.2503268

[41] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, 455–466.

[42] L. Mou, G. Li, L Zhang, T. Wang, and Z. Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293.

[43] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

[44] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[45] C. Mendis, C. Yang, Y. Pu, S. Amarasinghe, and M. Carbin. 2019. Compiler auto-vectorization with imitation learning. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*. Curran Associates, Inc., 14598–14609.

[46] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. In *Proceedings of the ACM Symposium on Programming Languages, Volume* 2. (OOPSLA), Article 147 (Oct. 2018), 25 pages. DOI: https://doi.org/10.1145/3276517

[47] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543.

[48] H. G. Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888.

[49] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, Piscataway, N. J., 404–415. DOI: https://doi.org/10.1109/ICSE.2017.44

[50] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[51] V. Raychev, M. Vechev, and A. Krause. 2015. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. ACM, New York, 111–124.

[52] N. Rosenblum, X. Zhu, and B. P. Miller. 2011. Who wrote this code? Identifying the authors of program binaries. In *Proceedings of the 16th European Conference on Research in Computer Security (ESORICS'11)*. Springer-Verlag, Berlin, 172–189. http://dl.acm.org/citation.cfm?id=2041225.2041239

[53] M. Stephenson and S. Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization*. 123–134. DOI: https://doi.org/10.1109/CGO.2005.29

[54] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic construction of inlining heuristics using machine learning In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. IEEE Computer Society, 1–12. DOI: https://doi.org/10.1109/CGO.2013.6495004

[55] Nicolai Stawinoga and Tony Field. 2018. Predictable thread coarsening. *ACM Trans. Arch. Code Optim.* 15, 2, Article 23 (June 2018), 26 pages. DOI: https://doi.org/10.1145/3194242

[56] I. Sutskever, O. Vinyals, and Q. V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, 3104–3112. http://dl.acm.org/citation.cfm?id=2969033.2969173.

[57] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE Press, Article 31, 11 pages.
[58] Martin Vechev and Eran Yahav. 2016. Programming with "big code ". *Found. Trends Program. Lang.* 3, 4 (Dec. 2016), 231–284. DOI: https://doi.org/10.1561/2500000028
[59] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan. 2016. Bugram: Bug detection with N-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, 708–719. DOI: https://doi.org/10.1145/2970276.2970341
[60] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems* 2, 1–3 (1987), 37–52.
[61] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic neural program embeddings for program repair. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018), (Vancouver, Canada, Apr 30 - May 3, 2018)*. https://openreview.net/forum?id=BJuWrGW0Z.
[62] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*. AAAI Press, 1112–1119.
[63] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 416–426. DOI: https://doi.org/10.1109/ICSE.2017.45

Received February 2020; revised July 2020; accepted August 2020