GRAPHSPY: Fused Program Semantic Embedding through Graph Neural Networks for Memory Efficiency

Yixin Guo* — *Peking University*, Beijing, China — yixinguo@pku.edu.cn
Pengcheng Li† — *Alibaba Damo Academy*, Sunnyvale, CA, USA — landy0220@gmail.com
Yingwei Luo — *Peking University*, Beijing, China; *Peng Cheng Lab*, Shenzhen, China — lyw@pku.edu.cn
Xiaolin Wang — *Peking University*, Beijing, China; *Peng Cheng Lab*, Shenzhen, China — wxl@pku.edu.cn
Zhenlin Wang — *Michigan Tech*, Houghton, USA — zlwang@mtu.edu

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586120

***Abstract*—Production software oftentimes suffers from unnecessary memory inefficiencies caused by inappropriate use of data structures, programming abstractions, or conservative compiler optimizations. Unfortunately, existing works often adopt a whole-program fine-grained monitoring method incurring incredibly high overhead. This work proposes a learning-aided approach to identify unnecessary memory operations, by applying several prevalent graph neural network models to extract program semantics with respect to program structure, execution semantics and dynamic states. Results show that the proposed approach captures memory inefficiencies with high accuracy of 95.27% and only around 17% overhead of the state-of-the-art.**

***Index Terms*—Dead store, GNN, program representation**

I. INTRODUCTION

Production software oftentimes suffers from various kinds of performance inefficiencies. Some inefficiencies are induced by programmers during design (e.g., poor data structure selection) and implementation (e.g., use of heavy-weight programming abstractions). Others are compiler induced, e.g., not inlining hot functions. Identifying and eliminating inefficiencies in programs are important not only for commercial developers, but also for scientists writing compute-intensive codes for simulation, analysis, or modeling. Performance analysis tools, such as *gprof* [1], *HPCToolkit* [2], and *vTune* [3], attribute running time of a program to code structures at various granularities (usage analysis). However, such tools are incapable of identifying unnecessary memory operations (wastage analysis).

Dead stores are one type of unnecessary memory operations. A dead store happens when two consecutive store instructions to the same memory location are not intervened by a load instruction for the same location. Fig. 1 shows an example in the `Hmmer` program from the SPEC CPU2006 benchmark suite. The source code first writes, then reads and finally writes `mc[k]`, showing no dead stores. But in the assembly code in Listing 2, the value of `mc[k]` is held in a register, which is reused (read) in the comparison on line 4. The memory location `0x4(%rsi)` is written first on line 2 and then on line 6. Hence, line 2 is a dead store.

Whole-program fine-grained monitoring is a means to monitor execution at microscopic details. It monitors each binary instruction instance, including its operator, operands, and run-time values in registers and memory. A key advantage of microscopic program-wide monitoring is that it can identify redundancies irrespective of the user-level program abstractions. Prior works [4], [5] have shown that the fine-grained profiling techniques can effectively identify dead stores and offer detailed guidance. However, microscopic analysis incurs an unacceptable cost in practical use. It is reported that the state-of-the-art (the best-paper awardee) takes up to 150× run-time slowdown [6].

```
1 mc[k] = mpp[k-1] + tpmm[k-1];
2 if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
3   mc[k] = sc;
```

Listing 1: Dead stores in Hmmer.

```
1 add (%rbx,%rcx,4),%eax #hoist the computation result in %eax
2 mov %eax,0x4(%rsi)     #the 1st store to mc[k]
3 ...
4 cmp %edx,%eax          #the conditional check uses mc[k]'s
    value stored in %eax instead of 0x4(%rsi)
5 ...
6 mov %edx,0x4(%rsi)     #the 2nd store to mc[k]
```

Listing 2: The assembly code of Listing 1.

Fig. 1. A dead store example in the `Hmmer` program.

To this end, we propose a learning-aided approach that assists in finding out the procedures that are highly likely to have dead stores and then employ the fine-grained monitoring tool for these suspicious targets, rather than whole-program fine-grained monitoring. As a result, our approach reduces the running time overhead of the state-of-art tool by up to 124×.

We formulate this prediction problem as a graph-level task. A procedure is represented by a graph. We embed a procedure through a graph neural network and feed the embedding to a classifier in the final stage to predict whether dead stores exist in the procedure. Unfortunately, the complexities of a procedure involving various syntactic and semantic factors, such as the control flow, data flow, input-sensitivity, context-sensitivity, and many architecture-specific instructions, bring an utmost challenge to program embedding.

In order to capture the intra-procedural structure information, we apply the gated graph neural network [7] (GGNN) onto control flow graphs (CFGs) through the message-passing neural network framework. However, intra-procedural structure information cannot envision dead stores caused across procedure boundaries. In order to make the prediction more precisely, inter-procedural structure information needs to be additionally embedded by dynamically profiling a calling context tree [8]. A calling context tree reveals run-time calling relations between procedures. We apply the GGNN onto it to capture inter-procedural structure information. Finally, we embed dynamic value semantics by taking snapshots of memory addresses and associated values stored during program execution, and combine the embedding with the inter-procedure graph embedding. The embedding of memory states enhances the prediction precision by encoding input-sensitivity, data dependency flow, and execution states of a program.

*The work was done when Yixin Guo was an intern at Alibaba.
†Corresponding author.

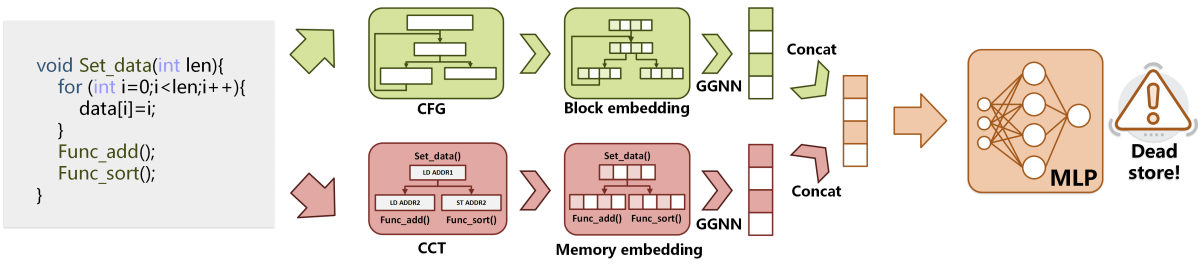


Fig. 2. Overview of the program representation using fused static and dynamic embeddings based upon GNNs.

In summary, this paper makes the following contributions:

- We present a novel approach called GRAPHSPY for detecting dead stores with low overhead. As far as we know, this is the first work that applies GNNs for dead store detection.
- We present a hybrid embedding approach that embeds both intra- and inter-procedural structure semantics with static CFGs and dynamic calling context trees.
- We present an embedding-based data flow graph that captures program value semantics by sampling memory and register states with negligible overhead.
- We evaluate GRAPHSPY on the SPEC CPU benchmark suite for a number of combinations of different compilation options and platforms. GRAPHSPY realizes the average prediction accuracy of 95.27%, with only 17% overhead of the state-of-the-art tool.

II. PROGRAM REPRESENTATION AND EMBEDDING

In this section, we will introduce in detail the approach to build network embeddings and how to utilize these embeddings to predict dead stores. Fig. 2 shows the diagram of the overview of the proposed approach using fused static and dynamic embedding.

A. Graph Neural Networks

The objective of Graph Neural Network is to learn the node representation and graph representation for predicting node attributes or attributes of the entire graph. A *Graph Neural Network (GNN)* [9] structure $G = (V, E)$ consists of a set of node V and a set of edge E. Each node $v \in V$ is annotated with an initial node embedding by $x \in \mathbb{R}^D$ and a hidden state vector $h_v^t \in \mathbb{R}^D$ (h_v^0 often equals to x). A node updates its hidden state by aggregating its neighbor hidden states and its own state at the previous time step. In total, T steps of state propagation are applied onto a GNN. In the t-th step, node v gathers its neighbors' states to an aggregation as m_v^t, as shown in (1). Then the aggregated state is combined with node v's previous state h_v^{t-1} through a neural network called g, as shown in (2). f can be an arbitrary function, for example a linear layer, representing a model with parameters θ.

$$m_v^t = \sum_{(u,v)\in E} f(h_v^t; \theta) \tag{1}$$

$$h_v^t = g(m_v^t; h_v^{t-1}) \tag{2}$$

Gated Graph Neural Network (GGNN) [7] is an extension of GNN by replacing g in (2) with the *Gated Recurrent Unit (GRU)* [10] function as shown in (3). The GRU function lets a node memorize long-term dependency information, as it is good at dealing with long sequences by propagating the internal hidden state additively instead of multiplicatively so that the temporal relations can be first propagated and then accumulated from node to node.

$$h_v^t = \mathbf{GRU}(m_v^t; h_v^{t-1}) \tag{3}$$

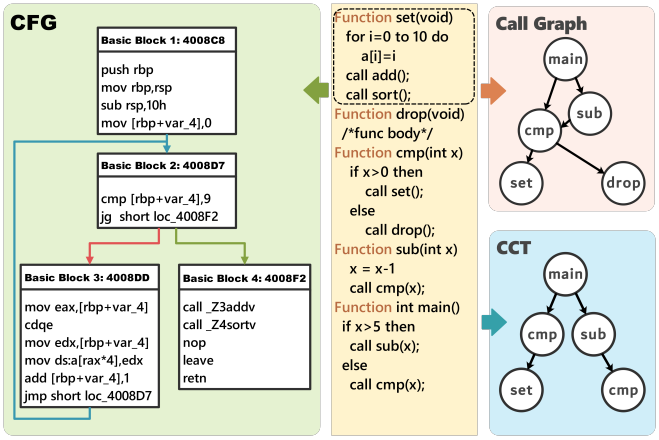


Fig. 3. Examples of a control-flow graph and the comparison between a call graph and a calling-context tree.

B. Static Intra-Procedural Structure Embedding

An *assembly code* can be compiled from a source code. An assembly code is specific to a processor architecture, for example, `x86-64`. It is a flat profile of *instructions*, $Ins_i (i \in 1 \dots m)$. Each instruction Ins_i is composed of a series of *tokens*, $t_i (i \in 1 \dots n)$. These tokens are of many types, including operators and operands. Some instructions are compute operations with respect to register values (e.g., `ADD`, `SUB`). Some move values between registers and memory (e.g., `LOAD`, `STORE`). Others represent conditional branch or jump to other locations (e.g., `JZ`, `JMP`).

A basic block $B_i (i \in 1 \dots K)$ is composed of a sequence of instructions without any control flow statements. A basic block has only one entry and one exit and may have multiple following basic blocks. When the program pointer jumps from a source to a target block, we connect an edge from the source to the target. Viewing the basic blocks as nodes and the connections as edges, we form a *control flow graph (CFG)*. For instance, for x86 direct branches, there are only two possible target blocks for a given source block, which we can refer to as the *true block* and *false block*. Fig. 3 shows a procedure named `set` and its according *CFG*. A *CFG* represents intra-procedure program structure semantic, so we attempt to embed a *CFG* into a feature map. We choose GGNN to build the embedding of the entire graph.

The semantics of instructions is crucial for identifying dead stores. By tokenizing every instruction, we treat a token as a word and an instruction as a sentence to take advantage of natural language models for embedding. First, we apply the word2vec [11] model to encode every token in an instruction into a fixed-length vector. The

embedding process by the word2vec model was introduced in the existing works [12]–[14]. Then we average out token embeddings to generate an embedding of an instruction, and finally derive a whole embedding of a basic block by averaging the embeddings of all instructions inside. Equation (4) depicts the embedding process.

$$\forall k, e_{bb_k} = \frac{\sum_{j=0}^{m} \{e_{Ins_j} = \frac{\sum_{i=0}^{n} \textbf{word2vec}(t_i)}{n}\}}{m} \quad (4)$$

Once we have embedded every basic block, we can build the embedding of the whole graph through GGNN. Since we embed it from the assembly without any actual executions, this type of embedding is static and takes no cost for the run-time execution.

Abstract Syntax Tree (AST) [15] is an intermediate representation of a binary generated in the process of compilation from the source code to binary code. It uses context-free grammar parsing rules that partially include the grammars and execution orders of a binary. By contrast, we choose to construct a *CFG* from the assembly code, because it is typically less stylish and tighter to program semantics. For example, programs that are syntactically different but semantically equivalent tend to correspond to similar assembly codes. Moreover, the assembly code often embraces more architecture-specific characteristics than AST so that the embedding encodes architecture level information which is critical to dead store detection.

C. Dynamic Inter-Procedural Calling-Relation Embedding

*1) CCT Profiles: CFG*s represent only intra-procedural structure semantic, rather than whole-program structure. *Call graph* captures inter-procedural structure information, i.e., caller-callee relations. Nevertheless, call graph only expresses static structure information that may not reflect the actual calling sequence of function calls. An alternative is to profile *calling context tree (CCT)*s [8], which is the actual function-call sequence during program execution.

In *CCT* profiles, data (e.g., function call) is collected with respect to each stack frame. *CCT*s comprise the merged call-stacks from all functions invoked during the course of program execution, with each frame in the stack represented as a node in the tree, and with common prefixes merged. Typically, each call stack in the *CCT* is rooted at `main`. A *CCT* can differentiate a function called in different ways from different contexts with different parameters and global states. Furthermore, when time measurement data is aggregated across threads, the aggregated time obscures problems occurring on a specific subset of *CCT*s.

Fig. 3 compares the *static* call graph and *dynamic* calling context tree constructed from the same code. In the upper call graph, there is only one instance for any single procedure and the caller-callee relations are formed statically, for example from `cmp(x)` to `set()`. While in the bottom *CCT* example, two instances of `cmp()` exist with different input actual parameters, one of which is called by `sub` and the other is called by `main`.

2) Dynamic Value Snapshots: In addition to the *CCT* profiles, we periodically take program snapshots during program execution which are sets of values that occur over the course of the execution. We call every program snapshot, i.e. a set of values, as a program memory state, which is defined as a set of general-purpose registers and memory addresses and associated values. Different than registers, memory addresses are indexed by a 64-bit integer and hence extremely widespread. In theory, we should include all values of registers and the entire memory address space for a program memory state. However, it takes unacceptable space and time cost to profile the entire memory address space. Instead, for a program snapshot, we record general-purpose register values and only memory values stored within this execution. The *CCT*s and memory states are lightly profiled with negligible overhead through a binary instrumentation tool called DynamoRIO [16].

We associate profiled memory states to every function instance in a *CCT* graph. The memory states are used to initialize the embedding of a node, i.e., a function instance, in a *CCT* graph. The memory states include the largest and smallest accessed addresses, number and unique number of memory accesses, number of accessed pages, and most frequently accessed pages. We concatenate these manually selected features into a vector and use it as the initial node embedding. After the initialization of node embedding, we run the GGNN model again on the *CCT* graph to propagate memory state semantics across procedure boundaries to formalize a *CCT* embedding.

Program memory states characterize memory value semantics, i.e., which functions load and store which memory addresses. Because dead stores may happen between two function instances, the GGNN propagation of memory states in the *CCT* aides in capturing these dead store cases. We call the embedding of a *CCT* as inter-procedural structure semantics and memory value semantics.

D. Fused Static and Dynamic Embedding

Putting together both aforementioned embeddings, a single final program embedding is derived for the target procedure (Equation (5)). The program embedding is obtained by concatenating the intra-procedural structure embedding g_{intra} and inter-procedural embedding g_{inter}. Finally, we run the embedding through an MLP to check if the target procedure exists dead stores (Equation (6)). Fig. 2 summarizes the overview of the entire approach.

$$g_{prog} = g_{intra} \parallel g_{inter} \quad (5)$$

$$g_o = \textbf{MLP}(g_{prog}) \quad (6)$$

When the target procedure is predicted to be suspicious, we apply the state-of-art fine-grained monitoring tool, `CIDetector` [6] to carefully scan the target procedure line-by-line. As a result, `CIDetector` outputs the memory locations where it guarantees that dead stores happen. In this way, GRAPHSPY gets rid of a number of procedures that do not have dead stores, therefore, a substantial amount of time and space overhead for checking these procedures may be saved.

III. EVALUATION

In this section, we answer the following questions:

- What is the prediction accuracy (effectiveness) by GRAPHSPY for any single machine and compilation configuration?
- What is the cross-platform prediction accuracy (effectiveness) of a unified model targeting different architectures and options?
- What is time and space overhead improvement (efficiency) in contrast to the state-of-the-art fine-grained monitoring tool?

A. Experimental Setup

Dataset, tools and architectures. The proposed approach is composed of two types of inputs, static assembly code and dynamic program profiles. In order to collect static assembly, we compile the source code for a binary through the `GCC` and `LLVM` compilers. A binary analysis tool called `angr` [17] is utilized to construct CFGs by taking as input the binary code and outputting a CFG for every single procedure. A simple profiling tool was implemented to dump calling context trees and memory states based on DynamoRIO [16]. In order to obtain the ground truth data, i.e., dead stores of all binaries, we made our best efforts to carefully implement the state-of-the-art fine-grained monitoring tool, `CIDetector` [6] based on DynamoRIO.

TABLE I
BENCHMARK STATISTICS.

Binary	#Functions	#BBs	Data Volume
502.gcc	9174	273226	71.77GB
505.mcf	67	799	2.26GB
508.namd	92	1199	460MB
510.parest	18694	275693	10.51GB
511.povray	1381	29388	274MB
520.omnetpp	7451	56217	4.08GB
523.xalancbmk	12613	145245	4.33GB
526.blender	36782	314951	31.91GB
538.imagick	1616	48157	355MB
541.leela	506	5751	57MB
544.nab	287	7030	259MB
557.xz	355	4685	813MB

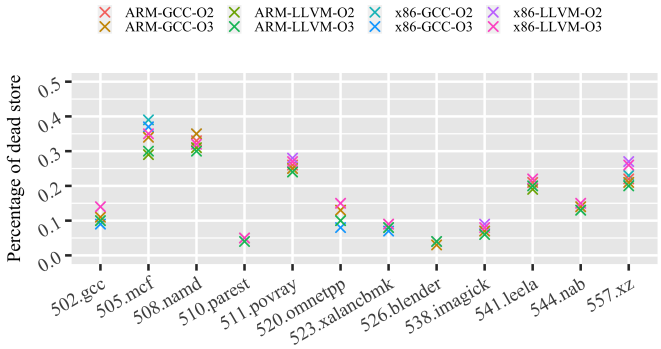


Fig. 4. Percentage of how many procedures have dead stores in a program.

TABLE II
DATASET SIZES OF DIFFERENT CONFIGURATIONS.

Config.	Training	Validation	Test	Total
x86-GCC-O2	19,427	4,221	4,221	27,869
x86-GCC-O3	18,451	4,154	4,154	26,759
ARM-GCC-O2	19,047	4,049	4,049	27,145
ARM-GCC-O3	18,524	4,040	4,040	26,604
x86-LLVM-O2	23,436	4,735	4,735	32,906
x86-LLVM-O3	23,063	4,754	4,754	32,571
ARM-LLVM-O2	19,247	4,055	4,056	27,358
ARM-LLVM-O3	19,019	4,078	4,078	27,175
Hybrid	159,719	34,084	34,085	227,888

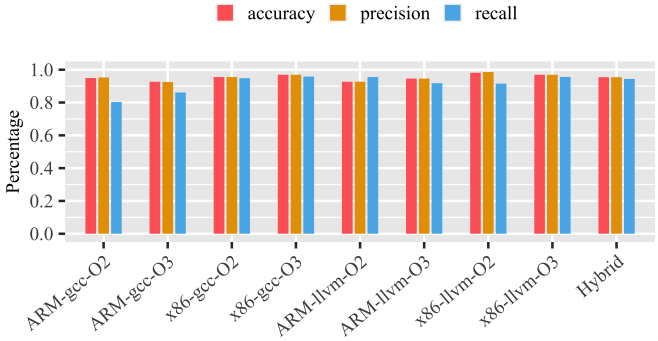


Fig. 5. Prediction results for different and hybrid configurations.

We attribute all found dead stores to every procedure of every binary. Each procedure corresponds to a data sample since we are predicting on a per-procedure basis. The procedure is labeled as 1 (true) if it has dead stores, 0 (false) otherwise. Fig. 4 shows the percentage of true samples over all samples for every program.

As is all known SPEC CPU benchmark suite [18] is a golden standard used to evaluate software system performance. We run programs from the benchmark suite with the reference input and sample program execution with for a few times. Table I shows the statistics of programs used, including the number of functions, number of basic blocks and, more importantly, raw data file volume that contains raw data of instructions, CFGs, memory values, CCT profiles, and label data.

We choose two micro-architectures, *x86-64* and *ARM* (AArch64), two compilers *GCC* and *LLVM*, two options *-O2* and *-O3* and compose eight configurations. We shuffle the samples from all binaries with any single configuration and take a fraction as training and testing datasets, as shown in Table II. The size ratio of true samples over false samples is around 1:1, for a balance of distribution.

Additionally, we create a hybrid configuration by mixing samples from eight basic configurations. This configuration is used to train a unified model that aims to work for any configuration. We also train our model with samples from the x86-GCC-O3 data set and test for samples from the other configurations, thus exhibiting good data diversity.

Hyper-parameters. We show only the values of all hyper-parameters used in the final experiments. The learning rate used is 0.0001 and the batch size is 256. The dimension of the word2vec embedding is 60 and the dimension of memory state embedding is 30. GGNN is used for both CFGs and CCTs. The output dimension and number of steps are 70 and 5 for CFGs, and 50 and 10 for CCTs.

Hardware. All deep learning tasks were performed on 8 Nvidia Tesla V100 GPU [19] cards of the Volta architecture. Each has 5120 streaming cores, 640 tensor cores and 32GB memory capacity. The CPU host is Intel Xeon CPU 8163 2.50GHz, running Linux kernel 5.0. All the other non-deep learning tasks were run on the host.

B. Prediction Results

Fig. 5 measures three metrics, precision, recall rate and accuracy for each configuration and the hybrid configuration. The precision metric is computed as $\frac{TP}{TP+FP}$, the recall rate as $\frac{TP}{TP+FN}$, and the accuracy as $\frac{TP+TN}{TP+FN+FP+TN}$, where TP denotes true positive, FP false positive, FN false negative, and TN true negative.

Clearly, all measurements are beyond 80%, which just confirms the efficacy of the proposed model for different configurations. On average, we achieve 95.38% of precision, 91.45% of recall rate and 95.27% of accuracy, respectively. The high prediction accuracy implies that GRAPHSPY is capable to rule out false samples for dead store detection. Hence, GRAPHSPY may save most of the checking overhead incurred by `CIDetector` for omitting those samples (i.e. procedures). We will show the time saving next.

It is noteworthy that we achieve high accuracy of 95.42% in the hybrid configuration. We trained the model with samples from different architectures and options, and tested for samples from different configurations as well. Drawing test samples from different configurations than train samples shows good data diversity. Although the GPU training time is within a few hours, results demonstrate that GRAPHSPY has great potential to be a unified model that performs well across different architectures and compilation options.

To further demonstrate the effectiveness of cross-platform, we trained the model with samples from the *x86-GCC-O3* data set, and tested for samples for the other configurations. Since train and test samples are drawn from different configurations, data diversity

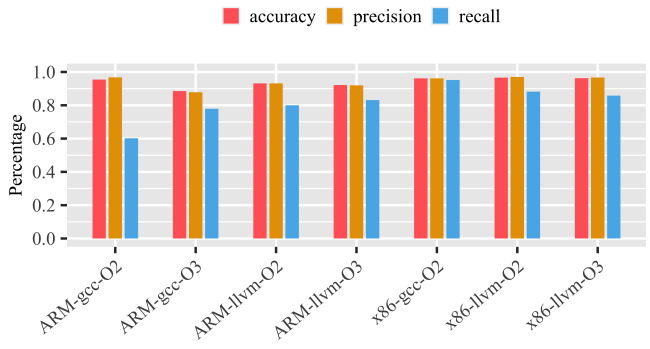


Fig. 6. Prediction results for the cross-configuration test.

is obtained. As shown in Fig. 6, the prediction results achieve consistently high accuracy over 90%. In this cross-platform test, test samples are drawn from different

C. Overhead Results

Taking *x86-GCC-O3* as an example, we measure time saving and memory overhead reduction by GRAPHSPY. We apply the model trained from the *x86-GCC-O3* data to find out dead stores for all binaries on *x86-GCC-O3*. `CIDetector` runs through a binary by filtering those procedures that do not have dead stores. Compared with whole-program monitoring, GRAPHSPY improves upon `CIDetector` by an average speedup of 5.79×, with a maximum of 25.35×, as shown by Fig. 7. Equivalently, GRAPHSPY takes only 17% overhead of `CIDetector`. Over the native run, GRAPHSPY incurs only the average slowdown of 25.9×, while `CIDetector` incurs around 150× slowdown. In the mean time, the memory cost incurred is reduced as well. Fig. 7 shows that GRAPHSPY reduces 65% of the memory overhead incurred by `CIDetector`.

D. Model Breakdown Results

We evaluate the efficacy of each component of the entire model used by GRAPHSPY upon two configurations *ARM-GCC-O2* and *ARM-GCC-O3*. Table III shows the recall/accuracy results. If only using the word2vec model, the baseline achieved prediction accuracy is 61.68% and 64.04%, respectively. Adding GGNN$_{intra}$ based upon word2vec, the accuracy is improved to 82.98% and 82.56%. Only using the GGNN$_{inter}$ model realizes poor accuracy, just 80.93% and 75.79%. When combining both GGNN$_{intra}$ and GGNN$_{inter}$, the maximum accuracy is obtained, 92.61% and 94.92%. This evaluation just confirms the necessity to combine both static intra-procedural structure embedding and dynamic inter-procedural value embedding.

We compared word2vec with the BERT [20] model in embedding semantic information of instructions. Nevertheless, incredibly high requirement of a large amount of data, unacceptable long training time, and tedious parameter tuning prevent us from achieving a highly effective BERT model and good prediction results. We only show the results by BERT with limited training data and time in Table III, which suggests that BERT is a hard-to-train model in practical use. Moreover, more models and networks can be experimented, but this is out of scope of this work.

IV. RELATED WORK

GNNs for program embedding. GNNs have attracted increasing attention to program representation, partly because many graph structures implicitly existing inside a program code, e.g., abstract syntax tree (AST), CFG and data flow graph, make GNNs highly applicable for program embedding [21]. Prior studies [22], [23] use the intermediate representation (IR) or AST to construct CFGs to feed to a gated graph neural network for program classification and data prediction tasks. Due to the inherent difference between syntax and semantics, models learned from static code can be imprecise at capturing semantic properties [24]. Our model constructs CFGs from the assembly code for a representation of program structure information. By it, architecture-specific details can be embedded by using instructions rather than generic intermediate representation.

Wang et al. [24], [25] embeds programs from dynamic execution traces or symbolic execution traces, which capture accurate program semantics, thus offering benefits that reason over syntactic representations. However, the quality of the model requires heavy execution profiling. Our dynamic model embeds dynamic program states from calling context trees and sampled memory addresses, which is very lightweight but brings sufficient semantics. Shi et al. [23] build graphs by taking instructions as nodes and value dependency between instructions as edges, and also takes program snapshots of memory values for dynamic embedding. However, it involves no any inter-procedural information and incurs overly much memory and computation overhead. Tal et al. [26] use LLVM IR to generate a representation that incorporates both the relative data- and control-flow, which is called conteXtual Flow Graph (XFG). They use XFG to train the embeddings for statement and apply it to a variety of program analysis tasks. Since this embedding method is hardware architecture independent, its ability to recognize dead stores is limited.

There are also several works learning from program representation to conduct program repair [24], [25], bug detection [27], program classification [22], cache replacement [28], [29], heap memory wise program verification [7], and code similarity detection [14]. Our work is the first to apply GNNs for dead store detection.

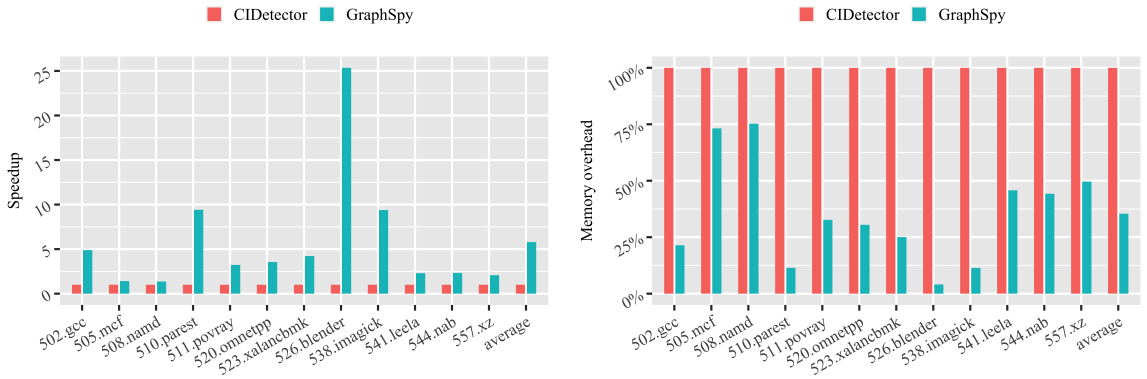
Memory wastage analysis. Wastage analysis focuses on how many computation and memory operations are unnecessary, for example, dead store detection [4], redundant load checking [6], run-time value numbering [30] and value locality exploration [5]. Unfortunately, existing checking tools have limited applicability because of as high as up to 150× overhead. Our work focuses upon dead store detection and improves the high overhead by a great margin.

TABLE III
MODEL BREAKDOWN.

Model	ARM-GCC-O2 Recall/Accuracy	ARM-GCC-O3 Recall/Accuracy
word2vec	0.6143/0.6168	0.5670/0.6404
GGNN$_{intra}$(word2vec)	0.6628/0.8298	0.7785/0.8256
GGNN$_{inter}$	0.9157/0.8093	0.9218/0.7579
BERT	0.6133/0.5740	0.6910/0.4580
GGNN$_{intra}$(BERT)	0.5489/0.7610	0.3066/0.7940
GRAPHSPY	**0.8613/0.9261**	**0.8033/0.9492**

V. CONCLUSION

In this paper, we have presented a new learning aided approach namely GRAPHSPY built upon graph neural networks to detect dead stores with reduced overhead. As far as we know, it is the first work that applies GNNs for dead store detection. We have designed a novel embedding approach that embeds intra- and inter-procedural structure semantics and dynamic memory value semantics to enhance the detection precision. The conducted evaluation suggests



(a) Time Speedup over CIDetector

(b) Memory Reduction over CIDetector

Fig. 7. Time saving and memory reduction over `CIDetector`.

that GRAPHSPY achieves as high as 95.27% of accuracy with only 17% overhead of the state-of-the-art.

ACKNOWLEDGMENT

We thank all anonymous reviewers for their insightful and helpful feedback. The research is supported in part by the National Key RD Program of China under Grant No. 2018YFB1003604, and the National Science Foundation of China (Nos. 62032001, 62032008, 61672053 and U1611461) and National Science Foundation CSR1618384.

REFERENCES

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler (with retrospective)," in *Best of PLDI*, pp. 49–57, 1982.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurrency and Computation : Practice Experience*, vol. 22, p. 685–701, Apr. 2010.

[3] "Intel VTune Amplifier XE 2013." http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe, 2013.

[4] M. Chabbi and J. Mellor-Crummey, "Deadspy: A tool to pinpoint program inefficiencies," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), p. 124–134, Association for Computing Machinery, 2012.

[5] S. Wen, M. Chabbi, and X. Liu, "Redspy: Exploring value locality in software," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[6] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant loads: A software inefficiency indicator," in *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[7] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *Proceedings of ICLR'16*, April 2016.

[8] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 85–96, 1997.

[9] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[10] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[11] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of wor representations in vector space," 2013.

[12] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *arXiv preprint arXiv:1806.07336*, 2018.

[13] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.

[14] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the National Conference on Artificial Intelligence and on Innovative Applications of Artificial Intelligence*, pp. 1145–1152, 2020.

[15] J. F. I. Neamtiu and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1–5, 2005.

[16] D. Bruening, "Dynamorio: Efficient, transparent, and comprehensive runtime code manipulation." https://dynamorio.org/, 2004.

[17] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[18] SPEC, "Spec cpu benchmarks." http://www.spec.org/benchmarks.html#cpu, 2017.

[19] Nvidia-Inc., "Nvidia tesla v100 gpu architecture." https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[21] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. A. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017.

[22] M. Lu, D. Tan, N. Xiong, Z. Chen, and H. Li, "Program classification using gated graph attention neural network for online programming service," *CoRR*, vol. abs/1903.03804, 2019.

[23] Z. Shi, K. Swersky, D. Tarlow, P. Ranganathan, and M. Hashemi, "Learning execution through neural code fusion," in *International Conference on Learning Representations*, 2020.

[24] K. Wang, Z. Su, and R. Singh, "Dynamic neural program embeddings for program repair," in *International Conference on Learning Representations*, 2018.

[25] K. Wang and Z. Su, "Blended, precise semantic program embeddings," PLDI 2020, (New York, NY, USA), p. 121–134, Association for Computing Machinery, 2020.

[26] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems*, pp. 3585–3597, 2018.

[27] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020.

[28] P. Li and Y. Gu, "Learning forward reuse distance," *ArXiv*, vol. abs/2007.15859, 2020.

[29] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," *arXiv preprint arXiv:2006.16239*, 2020.

[30] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," vol. 53, p. 332–347, Mar. 2018.