COMPUTER RESEARCH AND MODELING
2023 VOL. 15 NO. 1 P. 211–224
DOI: 10.20537/2076-7633-2023-15-1-211-224

КиМ

ENGINEERING AND TELECOMMUNICATIONS

UDC: 004.051

Performance prediction for chosen types of loops over one-dimensional arrays with embedding-driven intermediate representations analysis

R. K. Zavodskikh[a], N. N. Efanov[b]

Moscow Institute of Physics and Technology,
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia

E-mail: [a] roman.zavodskikh@phystech.edu, [b] nefanov90@gmail.com

Received 01.11.2022.
Accepted for publication 23.12.2022.

The method for mapping of intermediate representations (IR) set of C, C++ programs to vector embedding space is considered to create an empirical estimation framework for static performance prediction using LLVM compiler infrastructure. The usage of embeddings makes programs easier to compare due to avoiding Control Flow Graphs (CFG) and Data Flow Graphs (DFG) direct comparison. This method is based on transformation series of the initial IR such as: instrumentation — injection of artificial instructions in an instrumentation compiler's pass depending on load offset delta in the current instruction compared to the previous one, mapping of instrumented IR into multidimensional vector with IR2Vec and dimension reduction with t-SNE (t-distributed stochastic neighbor embedding) method. The D1 cache miss ratio measured with perf stat tool is considered as performance metric. A heuristic criterion of programs having more or less cache miss ratio is given. This criterion is based on embeddings of programs in 2D-space. The instrumentation compiler's pass developed in this work is described: how it generates and injects artificial instructions into IR within the used memory model. The software pipeline that implements the performance estimation based on LLVM compiler infrastructure is given. Computational experiments are performed on synthetic tests which are the sets of programs with the same CFGs but with different sequences of offsets used when accessing the one-dimensional array of a given size. The correlation coefficient between performance metric and distance to the worst program's embedding is measured and proved to be negative regardless of t-SNE initialization. This fact proves the heuristic criterion to be true. The process of such synthetic tests generation is also considered. Moreover, the variety of performance metric in programs set in such a test is proposed as a metric to be improved with exploration of more tests generators.

Keywords: mathematical modeling, compilers, intermediate representation, embeddings, performance analysis, static analysis

Citation: *Computer Research and Modeling*, 2023, vol. 15, no. 1, pp. 211–224.

КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2023 Т. 15 № 1 С. 211–224
DOI: 10.20537/2076-7633-2023-15-1-211-224

Ки&М

ИНЖИНИРИНГ И ТЕЛЕКОММУНИКАЦИИ

УДК: 004.051

Предсказание производительности избранных типов циклов над одномерными массивами посредством анализа эмбеддингов промежуточных представлений

Р. К. Заводских[a], Н. Н. Ефанов[b]

Московский физико-технический институт,
Россия, 141701, Московская область, г. Долгопрудный, Институтский пер., 9

E-mail: [a] roman.zavodskikh@phystech.edu, [b] nefanov90@gmail.com

Получено 01.11.2022.
Принято к публикации 23.12.2022.

Предложен метод отображения промежуточных представлений C-, C++-программ в пространство векторов (эмбеддингов) для оценки производительности программ на этапе компиляции, без необходимости исполнения. Использование эмбеддингов для данной цели позволяет не проводить сравнение графов исследуемых программ непосредственно, что вычислительно упрощает задачу сравнения программ. Метод основан на серии трансформаций исходного промежуточного представления (IR), таких как: инструментирование — добавление фиктивных инструкций в оптимизационном проходе компилятора в зависимости от разности смещений в текущей инструкции обращения к памяти относительно предыдущей, преобразование IR в многомерный вектор с помощью технологии IR2Vec с понижением размерности по алгоритму t-SNE (стохастическое вложение соседей с t-распределением). В качестве метрики производительности предлагается доля кэш-промахов 1-го уровня (D1 cache misses). Приводится эвристический критерий отличия программ с большей долей кэш-промахов от программ с меньшей долей по их образам. Также описан разработанный в ходе работы проход компилятора, генерирующий и добавляющий фиктивные инструкции IR согласно используемой модели памяти. Приведено описание разработанного программного комплекса, реализующего предложенный способ оценивания на базе компиляторной инфраструктуры LLVM. Проведен ряд вычислительных экспериментов на синтетических тестах из наборов программ с идентичными потоками управления, но различным порядком обращений к одномерному массиву, показано, что коэффициент корреляции между метрикой производительности и расстоянием до эмбеддинга худшей программы в наборе отрицателен вне зависимости от инициализации t-SNE, что позволяет сделать заключение о достоверности эвристического критерия. Также в статье рассмотрен способ генерации тестов. По результатам экспериментов, вариативность значений метрики производительности на исследуемых множествах предложена как метрика для улучшения генератора тестов.

Ключевые слова: математическое моделирование, компиляторы, промежуточные представления программ, эмбеддинги, анализ производительности, статический анализ

Introduction

Performance engineering is one of the actual topics in software development intended to search and resolve bottlenecks in architecture and implementation of components that lead to low performance of the entire software. Nowadays the expert estimation of dynamic profiling results is the most popular technique for performance analysis. Nonetheless, this approach has several disadvantages mainly related to the need to execute the program on bare metal or virtual environment which increases costs and makes it difficult at initial development stages. Therefore, the baseline performance estimation without execution gives a lot of possibilities for non-functional requirements improvement which make this task important.

This baseline performance estimation could be done on the top of compiler infrastructure which leads to the idea of extending the compiler with an expert module. This module should take a program as an input and automatically give some numerical characteristics as an output. Those characteristics should be correlated with an actual performance of the program taken by it. Programs with different semantics may have different performance even in the case where those numerical characteristics are close. Therefore the control-flows and data-flows also should be considered in this module. This work is focused on solving the problem described above using LLVM (Low-Level Virtual Machine) compiler infrastructure [Lattner, 2002; Lattner, Adver, 2004] and algorithms that can map programs to the vectors called embeddings [Makarov et al., 2021]. Embeddings are easier to compare as vectors of some, even implicitly obtained, vector space, unlike graphs, which at least should be tested on isomorphism or compared by some nontrivial metric to determine proximity [Foggia, Sansone, Vento, 2001; Makarov et al., 2021].

The performance prediction model could be created without using LLVM infrastructure, i. e. using dynamic sanitizer, but in this case the model would have no ability to gain information from control-flow graph which determines performance too. Moreover, we need to gain this information statically, not dynamically. Therefore, LLVM passes are used in this work to transform the C, C++ program into intermediate representation (IR) that is instrumented with performance estimation data. This IR could be transformed into embedding that contains information both about control-flow graph and performance estimation.

Related work

Software static performance estimation

Static software performance analysis has been an actively developing research area for several decades. During this period, several modeling approaches were proposed: execution graphs [Marie et al., 1997], stochastic Petri nets [Molloy, 1982], process algebras [Hermanns, Herzog, Katoe, 2002], queuing and layered queuing networks [Franks et al., 2009], UML-driven models [Garousi, Shahnewaz, Krishnamurthy, 2013]. Moreover, a list of approaches are proposed from case studies to generalize which architecture design of software is preferable under different usage scenarios, especially for distributed systems and web services [Trubiani et al., 2013; Bai et al., 2022; Maddodi, Jansen, Overeem, 2020]. Unfortunately, the main disadvantages of existing methods are crucial trade-offs between abstraction and complexity: an abstract model is easy to formalize and compute, but it will not reflect the real effects from environment, when a much more accurate model can be hard to construct and compute, due to the additional parameters involving, and not well-formalized corner-cases. It leads to continued search for relatively accurate and computationally efficient modeling approaches.

Embeddings-driven software analysis

One of the approaches that could be used for resolving trade-off between abstraction and complexity is embeddings usage because embeddings are easier to compare that control-flow and data-flow graphs and embeddings could also aggregate information from those graphs. However, embedding-driven representation analysis is a relatively new field of research. Recently, two classes of software embedding analysis approaches were proposed. The first approach is embeddings computation from intermediate representation, this approach is implemented in projects like IR2Vec [VenkataKeerthy et al., 2020] and [Zongjie et al., 2022]. The key idea of this approach is to construct a numerical vector called embedding from intermediate representation of program. In order to do this algorithm iterates over instructions presented in intermediate representation and aggregates information about those instructions to informative numerical vector. Such an algorithm could rely both on instruction types, like `load`, `store` or `add`, and instruction arguments like immediate data present in the instruction or memory address this instruction loads from or stores to. The second approach is embeddings construction directly from source code [Barchi et al., 2022; Alhoshan, Batista-Navarro, Zhao, 2019] using libraries like word2vec [Mikolov et al., 2013]. This approach considers a program source code as a sequence of tokens. Based on this assumption, the neural network from such a library which was tuned of a large corpus of similar texts (source codes in this case) takes the source code as an input and returns its embedding. Nevertheless, none of the proposed methods takes into account performance information or constructs models of memory usage by default. The modifications for the first approach which are aimed at taking performance information into account are considered in this article because they preserve control flow and data dependencies between instructions. This indicates the novelty of the proposed research, which, as the authors know, became one of the first works on embedding-driven representation analysis for static performance engineering.

Embedding-driven static performance estimation approach

Problem setting

Under the requirement of static performance estimation on the middle-end compiling phase, at first, the performance model which takes into account the information from IR structure should be defined. The most common way to do this is to provide the instrumentation of IR with model-specific counters containing performance information. Secondly, the comparison procedure between the instrumented IRs should be proposed and implemented for model calibration on real dynamic profiling data. On the one hand, comparison of IRs as graph-structured data through the graph isomorphism checking is computationally hard [Schoening, 1988; Bengoetxea, 2002]: in the general case the algorithm of checking whether two graphs are isomorphic is NP-hard. On the other hand, it takes $O(n)$ time to compute embeddings of both graphs with n vertices and $O(1)$ time to compare those embeddings. Therefore, the method of IRs mapping to vectors which saves the information about control and data-flows of the analyzed programs can be proposed: at first, the comparison of vectors in some latent vector-space is quite easy to compute, and then, if the information is enough to represent the structural properties of programs, the similarity of programs will be describable by the distance in vector-space.

The proposed method

Consider the program p^{test} from P set of test programs as an input translated into intermediate representation $IR(p^{test})$[1]. Let $Vec\colon P \to \mathbb{E}^n$ be a mapping, such as the distance between programs

[1] $IR(p^{test})$ could be achieved immediately from abstract syntax tree or after some passes performed to optimize the intermediate representation. This method could be used to evaluate performance after any pass.

in P is correlated with the distance between programs' embeddings in $\mathbb{E}^n$. Also, let a performance-descriptive numerical function $\Pi(p)$ be defined $\forall p \in P$. Then, "performance distance" between 2 programs $p_i, p_j \in P$ can be defined as $|\Pi(p_i) - \Pi(p_j)|$.

Following this consideration, the pipeline of programs analysis can be proposed:

1. Get $IR(p^{test})$ from the compiler.

2. Get $IR^i(IR(p^{test}))$ as an intermediate representation instrumented with performance information obtained statically using the performance model via injection of special variables and instructions. The method called AddVars described below which implements instrumentation using the basic reuse-distance cache miss statistics-related performance model is presented in this work and described in the next section.

3. Compute embedding $Vec\left(IR^i(IR(p^{test}))\right)$ and optionally provide dimensionality reduction. The IR2Vec and t-SNE [van der Maaten, Hinton, 2008] approaches described below are used in this article for embedding construction and dimensionality reduction (from $\mathbb{E}^n$ to $\mathbb{E}^2$), respectively. Intuitively, for the well-tuned embedding method, the programs with higher $\Pi(p)$ will mostly have a smaller $\mathbb{E}^2$ distance to another programs with higher $\Pi(p)$ that are injected with similar instructions/variables and have a huge $\mathbb{E}^2$ distance to programs with lower $\Pi(p)$ that are injected with another kind of instructions/variables.

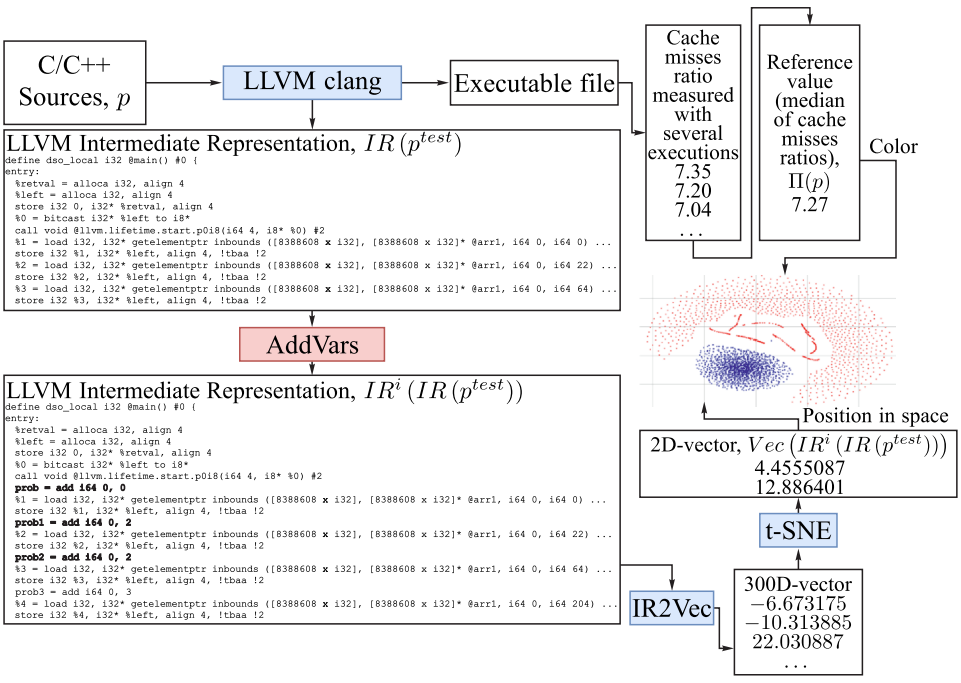

Figure 1. Proposed pipeline for programs performance analysis. C, C++ programs are compiled into executable files to measure the L1 cache miss ratio, on the one hand. On the other hand, the same programs are translated into LLVM IR, AddVars is used to inject artificial instructions, IR2Vec and t-SNE are used to create program embeddings after that

Therefore, in order to provide performance estimation, it is necessary to define some relevant numerical function $\Pi(p)$ and perform measurements for every program p in the train set. Consider the cache miss ratio as the relevant performance-descriptive numerical function $\Pi(p)$. The less cache ratio, the better because it means that the program tends to access the data already presented in cache-memory

more often than the data that needs to be achieved by slower RAM accessing. For every program p cache miss ratio measurement should be performed. This measurement can be done under Linux using perf stat [Becker, Chakraborty, 2018] tool which collects specific events occurring, while program execution via system counters reading, especially load cache hits and misses counters. For the purity of experiment, we use static CPU core affinity for the investigated program to avoid unpredictable inter-core migrations. Then, the correlations between $\mathbb{E}^n$ distance and "performance distance" for any pair of programs could be used to provide the cache miss ratio estimation criterion: if the cache miss ratio of program $\Pi\left(p^i\right)$ is known, the cache miss ratio of the other program $\Pi\left(p^j\right)$ should be estimated and $\left\|Vec\left(IR^i\left(IR\left(p^i\right)\right)\right) - Vec\left(IR^i\left(IR\left(p^j\right)\right)\right)\right\|$ is known to be small, then $\Pi\left(p^j\right)$ could be estimated close to $\Pi\left(p^i\right)$.

The frameworks used

The instumentation pass

LLVM is an infrastructure of compiler-related instruments consisting of modules for compilation, analysis and optimization of programs on multiple languages — C, C++, ObjectiveC, Fortran, etc. As well as any modern compiler, LLVM supports optimizing and analysis passes over the intermediate representation. Those passes are executed in predefined order at the middle-end stage, when the input program already parsed and transformed into mostly language-agnostic and architecture-agnostic intermediate representation called LLVM IR. Middle-end passes transform LLVM IR according to their inner logic. For example, constant folding pass folds all expressions with several constants into a single constant and loop unrolling pass creates a sequence of instructions instead of loop. Those standard passes are intended to decrease the execution time of program or its size. LLVM provides API for creating custom passes [Writing an LLVM Pass, 2022]. Moreover, it is obviously possible to serialize LLVM IR of any program to use it without LLVM infrastructure. For example, IR2vec [VenkataKeerthy et al., 2020; Jain et al., 2022] is an example of third-party analysis pass that transforms LLVM IR into a numerical vector called embedding.

AddVars pass is the LLVM pass developed while working on this article for instrumenting the analyzed programs. It passes every function presented in LLVM IR, for every function it iterates over instructions and for every load instruction it estimates the distance between memory addresses which is used by current load instruction and previous load instruction inside the same function. Depending on this difference, fictive instruction `%prob = add i64 0, some_number` is inserted. `%prob` is not used in this program ever more but `some_number` value depends on the adresses' difference as finite-valued step-wise function, given by Table 1.

For each of eight possible fictive instructions its base embedding should be defined which is different from base embeddings of all other instructions of the instruction set. This is needed for those fictive instructions to change the embedding of the entire program. It should be noticed that `%prob` is not used anywhere else, except in further embedding construction, and therefore the injected fictive instructions do not change the program semantics.

IR2Vec

IR2Vec [VenkataKeerthy et al., 2020] is the LLVM-oriented framework which generates vector representations from input programs in an unsupervised manner. The framework can be used to generate vector representations of programs processed by the compiler for further analysis[1]. The

[1] Static performance prediction task given in this paper also can be the example of such embedding-driven analysis.

Table 1. The values inserted in fictive instruction and the cases that correspond to those values. The case depends on the load address difference d between the current load instruction and the previous load instruction. In all cases when the difference is possible to estimate the more this distance the more the value inserted into artificial instruction

Value inserted	Ranges of d
1	$d \leqslant 8$ bytes
2	$8 < d \leqslant 64$ bytes
3	$64 < d \leqslant 512$ bytes
4	$512 < d \leqslant 4096$ bytes
5	$4096 < d \leqslant 16{,}384$ bytes
6	$16{,}384 < d \leqslant 65{,}536$ bytes
7	$65{,}536 < d \leqslant 524{,}288$ bytes
0	All other cases including the impossibility to estimate the difference at the compiling stage

IR2Vec framework uses LLVM IR programs as an input and maps those programs, by default, to 300-dimensional vectors called embeddings. Those embeddings aggregate the information from control-flows and data-flows, providing its representations as vector-valued “features”.

IR2Vec provides IR handling in two modes — Flow-Aware and Symbolic. Symbolic mode is a relatively naive approach to construct an embedding considering programs as token sequence regardless of control-flow graph and data-flow graph. Flow-Aware mode is more fine-tuned compared to the Symbolic because the output embedding of this mode aggregates the inter-instruction dependencies as well: the embedding of the separate instruction depends on all reaching definitions of the variables used in this instruction. The Flow-Aware mode of IR2Vec is considered to be used in this work because the instrumentation pass described above populates the IR by flow-aware information, which reflects cross-instruction data dependencies and memory-access patterns, thus, the main hypothesis that results from dynamic performance profiling will be well-correlated with the distances in embedding spaces can be checked on practice using this method. Moreover, IR2Vec is well-probated in a list of previous works [VenkataKeerthy et al., 2020; Jain et al., 2022] not related to performance estimation but relatively focused the programs scoring using distance estimation between embeddings.

t-SNE

t-SNE [van der Maaten, Hinton, 2008] is a statistical method for nonlinear dimensionality reduction usually used to map high-dimensional space vectors to two- or three-dimensional space for further visualization. This method was chosen because two-dimensional vectors are easier to use that high-dimensional ones, on the one hand, and this method maps vectors having the small distance between each other in high-dimensional space to the vectors having small distance between each other in the low-dimensional space.

There are two stages used in this algorithm. The first stage creates the probability distribution for every pair of vectors in high-dimensional space in such a way that vectors having small distance between each other will be assigned with high probability and vectors having huge distance between each other will be assigned with low probability. After that this method defines the same probability distribution for the corresponding vector in the low-dimensional space and minimizes Kullback – Leibler divergence (KL divergence) between two distributions varying the vector in the low-dimensional space.

While performing the experiment, attempts were made to use PCA [Jolliffe, Cadima, 2016] and t-SNE in the above-mentioned pipeline. The PCA usage does not yield any positive results

because, generally speaking, this dimensionality reduction method does not preserve distances between corresponding vectors. However, t-SNE does preserve those distances as well as UMAP [McInnes et al., 2018]. The results that could be obtained using UMAP instead of t-SNE are one of good tasks for the future research.

Experiment setting

The program set population

Public test sets [Rice et al., 2006] have a number of programs that is not enough for the proposed pipeline to be well-tuned. Therefore, as a proof of the concept for the proposed approach, the method of synthetic testing was chosen to be able to generate as huge variety of programs as required for making a conclusion based on statistical analysis of calculated embeddings: the more programs will be generated, the more representable performance estimation the above-mentioned pipeline can make. To create the experimental setup, at first, a huge enough set P of C programs described above should be generated. For simplicity of comparison the effects in model estimation with dynamic profiling, the programs that implement a single loop over the one-dimensional array is considered as target programs for tests population. Two sets of such programs are constructed, the "uniform" and "random distances" sets. Each set contains the programs with the same control flow, but different order of memory accessing. The generalized memory accessing policy for each set is related to the expected performance properties and is described below.

The first version of the program set generation script is called "uniform set" [Zavodskikh, 2022a]. The C, C++ line of code that was automatically generated is called command. Every program from the "uniform set" does several `load`s from the same array with different offsets. On every iteration, the offset is increased by cache size and after that the offset is taken by modulo $m = 1 + 2i$ where i is the number of the program in the "uniform set". The variety of m values causes the variety of space locality in programs from "uniform set": in the case where $i = 0$ the program always loads from offset equal to zero, in the case where i is a large, the program never loads from the same offset twice.

This script generated its own version of P set of programs, but in the "uniform set" it was impossible to rank programs by their cache miss ratio using programs' embeddings. This disadvantage was overcome using the second code generator script which generates "random distances set". The script is also available by the corresponding link [Zavodskikh, 2022b]. This script generate programs which iterates over a slice of array sequentially. On every iteration the program stores on the current element of array the sum of four elements: the first of them has a predetermined distance from the current element called i, three others have random distances:

- the second has the distance in range $\left[i, \frac{7}{8}i\right]$.
- the third has the distance in range $\left[i, \frac{1}{2}i\right]$.
- the fourth has the distance in range $[i, 0]$.

Every possible distance in given ranges has the same probability. The random differences make programs in "random distances set" more variable in the sense of memory reference distances and therefore in the sense of cache-efficiency.

Experimental results

To analyze the efficiency of every program generator, it is needed to measure dynamically the cache miss ratio with satisfactory precision in target scenarios on execution environment compatible with the proposed memory model using Linux Perf [Brendan, 2022]. After that we need to analyze the distribution of cache miss ratio in generated programs for every of two sets presented above. The more variable those ratios are the better, because it means that the model will be trained on a more broad set of programs and would be less likely overfitted. The next thing to check here is the fact that programs having similar embeddings have similar performance metric. Moreover, we should also take into account distributions of embeddings to check whether programs ordered by the cache miss ratio would also be ordered in some way in the embeddings space.

Let us consider the results from running experiments with the sets described in the previous subsection. The first one is the "uniform set". The script for "uniform set" generation was executed with parameters `generate_programs.py 8388608 2048 900`. From Fig. 2 the distribution of cache miss ratios that does not have significant spikes is illustrated. However, the number of commands which equals 2048 in generated programs is too huge to compute embeddings so it is impossible to say anything about embeddings' distribution in this case.

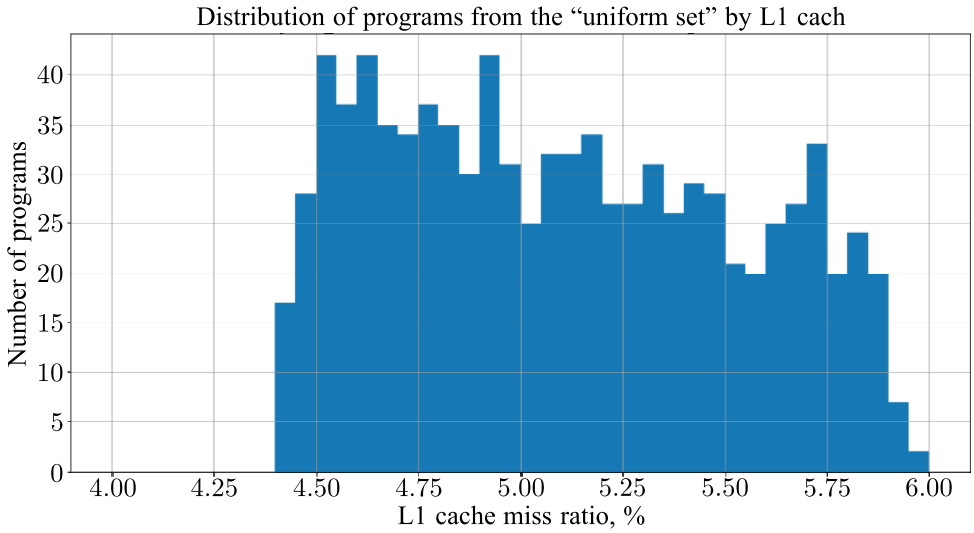

Figure 2. Distribution of median cache miss ratios in the "uniform set" with 2048 commands. Such a set of programs is called "uniform" because this distribution looks more or less uniform, it has no noticeable peaks

To simplify the embeddings computation it was decided to execute "uniform set" generator one more time with other parameters `generate_programs.py 8388608 512 900` to decrease the number of load instructions in programs to 512. In this case the distribution presented in Fig. 3 looks less various but has no significant spikes. For the "uniform set" the set of embeddings is presented in Fig. 4. This set of embeddings does not look informative, but it is still possible to divide programs that are less efficient from the programs that are more efficient.

In order to make this distribution more informative, the "random distances set" of programs was created. In Fig. 5 the distribution of cache misses ratio with spike is visualized. The corresponding distribution of embeddings presented in Fig. 6 represents a figure that looks like a continuous curve. Moreover, there is a high correlation between the cache miss ratio measured and the position of the embedding on the curve.

The correct work of this method could also be justified with correlation between the cache miss ratio and the distance between embedding of the current program and embedding of the worst program.

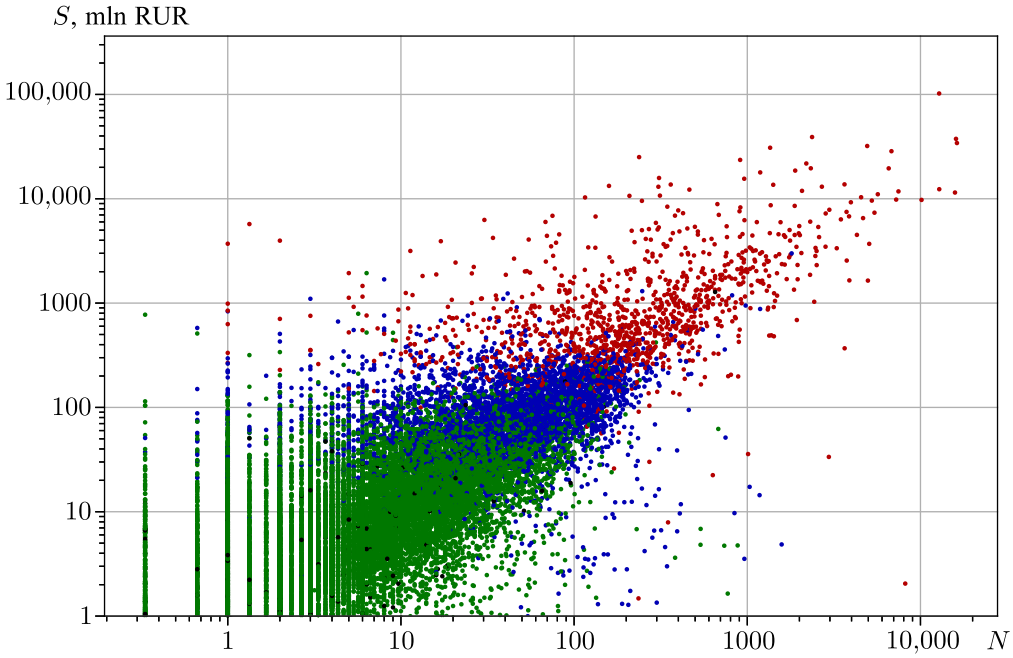

Figure 3. Distribution of median cache miss ratios in the “uniform set” with 512 commands. Such a set of programs is called “uniform” because this distribution looks more or less uniform, it has no noticeable peaks. The number of commands was reduced to be able to compute programs’ embeddings

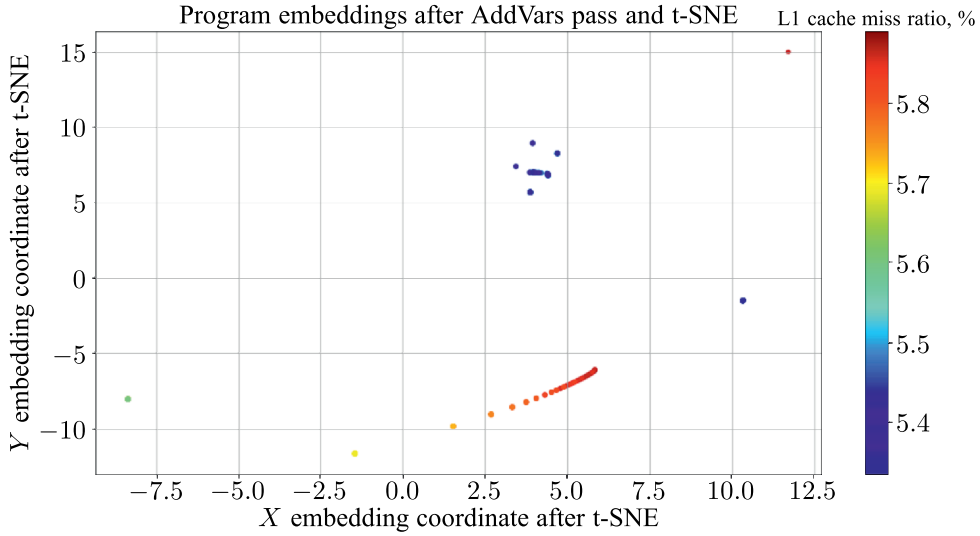

Figure 4. Embeddings of programs from the “uniform set”. The red ones represent less cache-efficient programs, when the blue ones represent more cache-efficient. The X and Y axes do not have a specific meaning because this figure is the output of the t-SNE algorithm

The value of such a correlation is not constant due to random initialization of t-SNE [van der Maaten, Hinton, 2008], however, we can compute a confidence interval for this value. As we can see from Table 2, there is a meaningful correlation between the distance from the worst program embedding and the cache miss ratio from the “uniform set” and the “random distances set”. This correlation is negative because the more the distance to the worst program, the more the probability of the program

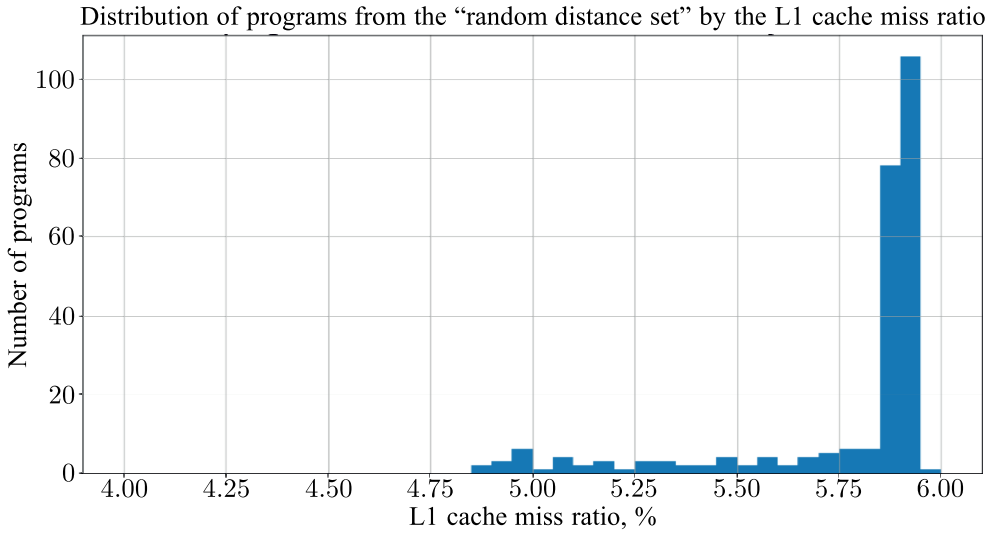

Figure 5. The distribution of median cache miss ratios in the “random distances set” of programs. This set is called “random distances set” because the offsets generated in programs are random. This distribution has a noticeable peak because programs generated in this set tend to be less cache-efficient

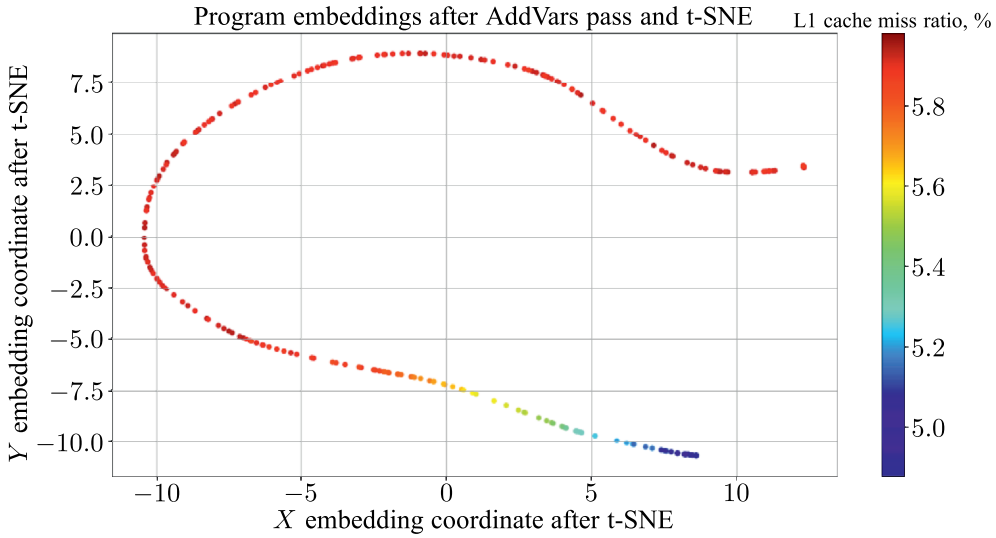

Figure 6. Embeddings of programs from the “random distances set”, lying on the 2-D curve. The position on the curve represents the cache miss ratio of the program (color-map from blue to red). The X and Y axes do not have a specific meaning because this figure is the output of the t-SNE algorithm

being “efficient”. So the more the distance to the worst program, the less the performance metric under consideration.

Table 2. The confidence intervals of correlation between the distance from the worst program embedding and the cache miss ratio for every set presented in this paper. Correlations are negative because the more the distance to the worst program embedding, the less cache miss ratio program tends to have

Set name	Correlation	Cache miss ratio range
Uniform	−0.609 ± 0.244	[5.3 %; 5.8 %]
Random distances	−0.607 ± 0.029	[4.9 %; 6 %]

This fact can be interpreted as a heuristic criterion to estimate the cache miss ratio for any investigated program p^{est} from the P set: having the above-mentioned pipeline initialized and trained, it is possible to calculate the LLVM IR of this program called $IR(p^{est})$, inject this LLVM IR with artificial instructions using AddVars to get $IR^i(IR(p^{est}))$ and finally compute its embedding $Vec\left(IR^i(IR(p^{est}))\right)$. The more the distance of this embedding to the embedding of the worst program from P set, the better the performance of p^{est} program.

The results indicate that, firstly, the instrumentation of LLVM IR with additional information about the difference of memory references could be used as the framework for compiling-time software performance modeling and, secondly, the method of IR embedding statistical analysis is practically suitable for static performance estimation from the constructed model.

Conclusion and future work

To conclude, the results of experimental evaluation provided above demonstrates the above-mentioned heuristic criterion to be practically suitable if the performance properties are mostly statically defined. This criterion can be used for performance estimation according to the predefined performance model on the chosen types of loops over one-dimensional arrays without direct CFGs and DFGs comparison. This direct comparison is avoided due to usage of embeddings constructed from IR.

Nevertheless, all above-mentioned program sets P were generated synthetically, therefore the next steps of research should be performed and all of them require experiments provided with a wide set of scenarios: refinement and improvement of the cache-memory model from AddVars pass, creating tests with more various cache miss distribution to avoid overfitting, developing new types of tests covering a larger set of programs beyond loops over one-dimensional arrays, evaluating the results on real programs. One more step to add here is tuning of test generators, IR2Vec and performance metric for increasing the modulo of correlation between the distance from the worst program embedding and the performance metric.

Moreover, performance of embeddings calculation itself is also an important research direction, because we had some scalability problems with P generation during the experiment despite highly-optimized Eigen [Eigen is a C++ template library for linear algebra. . . , 2022] library used inside IR2Vec. Also, we would like to consider other methods for dimensionality reduction instead of t-SNE, i. e., UMAP [McInnes et al., 2018].

References

Alhoshan W., Batista-Navarro R., Zhao L. Semantic Frame Embeddings for Detecting Relations between Software Requirement // Proceedings of the 13th International Conference on Computational Semantics. — Student Papers. — 2019. — P. 44–51.

Bai J., Chang X., Machida F., Han Z., Xu Y., Trivedi K. S. Quantitative understanding serial-parallel hybrid sfc services: a dependability perspective // Peer-to-Peer Networking and Applications. — 2022. — Vol. 15. — P. 1923–1938.

Barchi F., Parisi E., Bartolini A., Acquaviva A. Deep Learning Approaches to Source Code Analysis for Optimization of Heterogeneous Systems: Recent Results, Challenges and Opportunities // Journal of Low Power Electronics and Applications. — 2022. — Vol. 3. — 37 p.

Becker M., Chakraborty S. Measuring Software Performance on Linux, Technical Report // Technical University of Munich. — 2018.

Bengoetxea E. Inexact Graph Matching Using Estimation of Distribution Algorithms // Ecole Nationale Supérieure des Télécommunications, PhD Thesis. — 2002.

Brendan G. Perf Examples // 2022. — Available at: https://www.brendangregg.com/perf.html (accessed: 18.11.2022).

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms // 2022. — Available at: https://eigen.tuxfamily.org/index.php (accessed: 28.10.2022).

Foggia P., Sansone C., Vento M. A performance comparison of five algorithms for graph isomorphism // 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition. — 2001.

Franks G., Al-Omari T., Woodside M., Das O., Derisavi S. Enhanced Modeling and Solution of Layered Queueing Networks // IEEE Transactions on Software Engineering. — 2009. — Vol. 35. — P. 148–161.

Garousi V., Shahnewaz, S., Krishnamurthy D. UML-Driven Software Performance Engineering: A Systematic Mapping and Trend Analysis // Progressions and Innovations in Model-Driven Software Engineering. — 2013. — P. 18–64.

Hermanns H., Herzog U., Katoe J.-P. Process algebra for performance evaluation // Theoretical Computer Science. — 2002. — Vol. 274. — P. 43–87.

Jain S., Andaluri Y., VenkataKeerthy S., Upadrasta R. Poset-RL: Phase ordering for optimizing size and execution time using reinforcement learning // 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). — 2022. — P. 121–131.

Jolliffe I. T., Cadima J. Principal component analysis: a review and recent developments // Philosophical Transactions of the Royal Society A. — 2016.

Lattern C., Adver V. LLVM: A compilation framework for lifelong program analysis transformation // CGO'04: Proceedings of the international symposium on Code generation and optimization. — 2004. — P. 75–86. — http://portal.acm.org/citation.cfm?id=977395.977673&coll=GUIDE&dl=GUIDE&CFID=48424181&CFTOKEN=16724426 (accessed: 28.10.2022).

Lattner C. LLVM: An Infrastructure for Multi-Stage Optimization // Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign. — 2002. — http://llvm.cs.uiuc.edu

Maddodi G., Jansen S., Overeem M. Aggregate Architecture Simulation in Event-Sourcing Applications Using Layered Queuing Networks // Proceedings of the ACM/SPEC International Conference on Performance Engineering. — 2020. — P. 238–245.

Makarov I., Kiselev D., Nikitinsky N., Subelj L. Survey on graph embeddings and their applications to machine learning problems on graphs // PeerJ Computer Science. — 2021. — Vol. 7. — P. 357.

Marie R. A., Plateau B., Calzarossa M., Rubino G. Computer Performance Evaluation: Modelling Techniques and Tools // International Conference, St. Malo, France. — 1997. — Vol. 1245.

McInnes L., Healy J., Saul N., Grossberger L. UMAP: Uniform Manifold Approximation and Projection // The Journal of Open Source Software. — 2018. — Vol. 3, No. 29. — P. 861.

Mikolov T. Chen K., Corrado G., Dean J. Efficient Estimation of Word Representations in Vector Space // 2013.

Molloy M. Performance analysis using stochastic Petri nets // IEEE Transactions on Computers. — 1982. — Vol. C-31. — P. 913–917.

Schoening U. Graph isomorphism is in the low hierarchy // Journal of Computer and System Science. — 1988. — Vol. 37, No. 3. — P. 312–313.

Rice D., Biller L., Glick J., Sandifer C., Valencia B. SPEC CPU 2006 // Standard Performance Evaluation Corporation. — 2006. — Available at: https://www.spec.org/cpu2006/ (accessed: 11.11.2022).

Trubiani C., Meedeniya I., Cortellessa V, Aleti A., Grunske L. Model-Based Performance Analysis of Software Architectures under Uncertainty // Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures. — 2013. — P. 69–78.

van der Maaten L., Hinton G. Visualizing Data using t-SNE // Journal of Machine Learning Research. — 2008. — Vol. 9, No. 86. — P. 2579–2605.

VenkataKeerthy S., Aggarwal R., Jain R., Desarkar M. S., Upadrasta R., Srikant Y. N. IR2VEC: LLVM IR Based Scalable Program Embeddings // ACM Transactions on Architecture and Code Optimization. — 2020. — Vol. 17, No. 2. — 27 p.

Writing an LLVM Pass. — [Electronic resource]. — Available at: https://llvm.org/docs/WritingAnLLVMPass.html (accessed: 07.05.2022).

Zavodskikh R. 1st code generator script. — [Electronic resource]. — Available at: https://github.com/RomanZavodskikh/ PhD/blob/master/code_generator2.py (accessed: 18.11.2022a).

Zavodskikh R. 2nd code generator script. — [Electronic resource]. — Available at: https://github.com/RomanZavodskikh/ PhD/blob/master/code_generator3.py (accessed: 18.11.2022b).

Zongjie L., Pingchuan M., Huaijin W., Shuai W. Unleashing the power of compiler intermediate representation to enhance neural program embeddings // ICSE'22: Proceedings of the 44th International Conference on Software Engineering. — 2022. — P. 2253–2265.