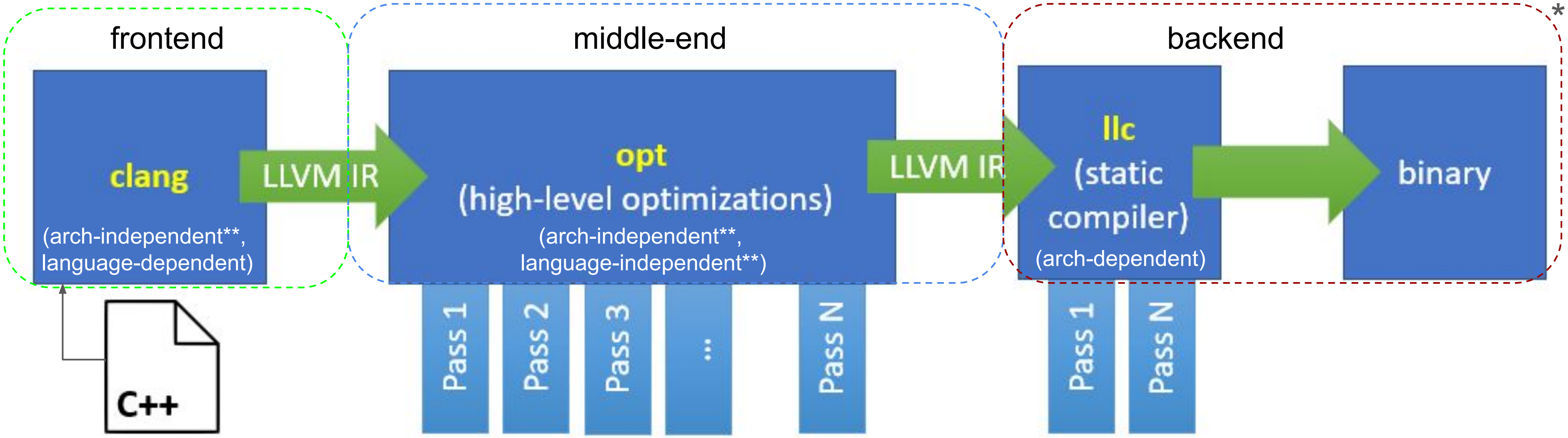
Compiler Auto-tuning:
Automatic search of optimal options / pass sequences / parameters



Nikolay Efanov, PhD
Associate Professor at MIPT

* Clang / LLVM
** There are some nuances

Compiler auto-tuning:

Compilers implements a lot of different transformation passes

- LLVM: ~160
- GCC: ~250
- At all, there about ~500 different types of passes
- Some passes are parametric (“loop-unroll”, etc.)

Pass sequences, which are “optimal in mean” for different optimization targets (size, perf, calculation accuracy), are grouped into “optimization levels”:

- O0, O1
- O2 (usually used in industry)
- O3
- Os, Oz…

Passes have different granularity levels, and require different program representations

Some programs (mostly all, in fact) can be better optimized, than -O*, if to tune the passes and their parameters more accurately.
The sequence of passes also oftenly important (AB != BA).

Problems and combinatorial estimations of search spaces

- Choosing the right set of phases:

$$\boxed{|\Omega_{selection}| = \{0,1\}^n} \tag{1}$$

(1) Estimates the search space size for binary selection of phases. For example, if $n = 9$ then $|\Omega_{selection}| = 2^9 = 512$. Moreover, if taking into account parameterization of phases and phases repetition (some of phases can be applied more than 1 time), the extended estimation:

$$|\Omega_{selection\_extended}| = \prod_{j=1}^{n'} \{0, 1, ...m_j\}, where \tag{2}$$

$m_j + 1$ is the total number of parameters' values for $j-th$ phase, $j \in [1, n']$, and n' is number of phases to apply.

- Phase-ordering problem: Due to the permutations, the search space size of this problem growth as factorial (3):

$$\boxed{|\Omega_{phases}| = n!} \tag{3}$$

Moreover, taking into account possible repetitions of phases and variability of sequence length, this estimation becomes (4):

$$|\Omega_{phases\_extended}| = \sum_{i=0}^{l} n^i, where \tag{4}$$

l is the max length of phases sequence vector.

Thus, there is, for example, 3905 variants for *l=5*

Note 1:
Methods for automation of phases choosing, phase-ordering and phase parameters picking required
Note 2:
There is no “ideal” order of passes applying in common case. Pass A transforms code in that it can corrupt the optimizations, which could be successfully performed by the pass B next to the A.

Compiler auto-tuning: passes reordering

Different objectives, pass sets, even constrains:

Objective function

Constrains

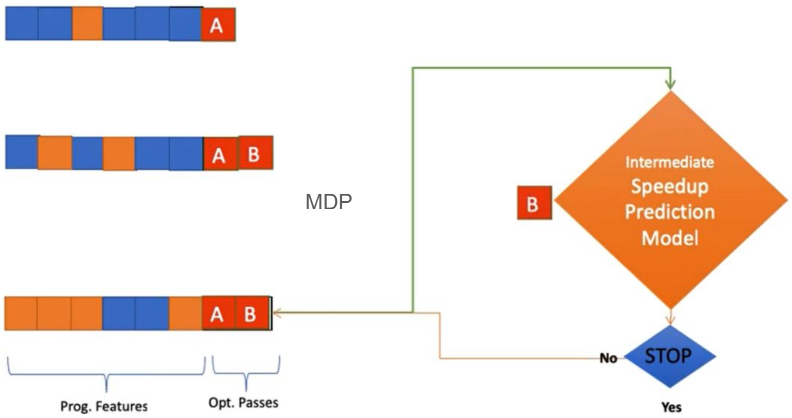
For whole program p granularity

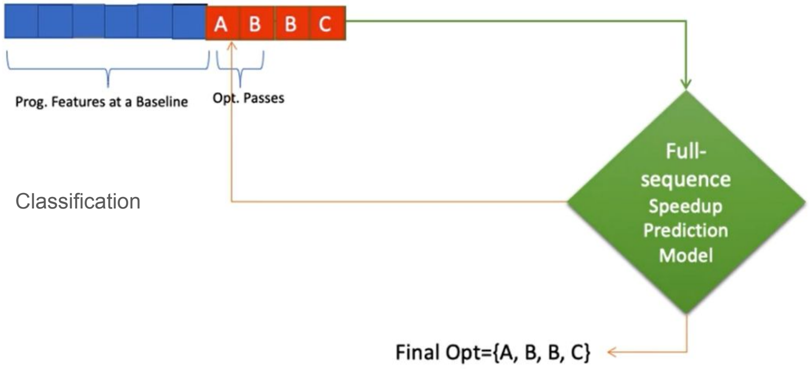
$$\boxed{size(C(s,p))}\ \boxed{\Delta rt(C(s,p),T,O2) < \varepsilon} \rightarrow min\text{, где}$$

For function-level granularity

$$\boxed{\sum_{i=1}^{|F(\varepsilon)|} \widehat{size(C(s_i, f_i))}}\ \boxed{\Delta rt(C(s_i,f_i),T,O2) < \varepsilon} \rightarrow min$$

Different solution approaches:





1. Iterative (n passes are applied, predict n+1)
   - \+ Models can be relative simple
   - \- One characterization per step
   - \- Local min convergency risks

2. Full-sequence prediction
   - \- Models are more complicated
   - \- Requires accurate program representations
   - \+ Requires only one characterization on inference
   - \+ Much resistance to local min

Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Trans. Archit. Code Optim. 14, 3, Article 29 (September 2017), 28 pages. https://doi.org/10.1145/3124452

- **Support by compiler**
  - Switching passes on / off (in fixed order)
  - Pass-reordering
  - Infrastructure for working with IR, representations, optimizer, etc

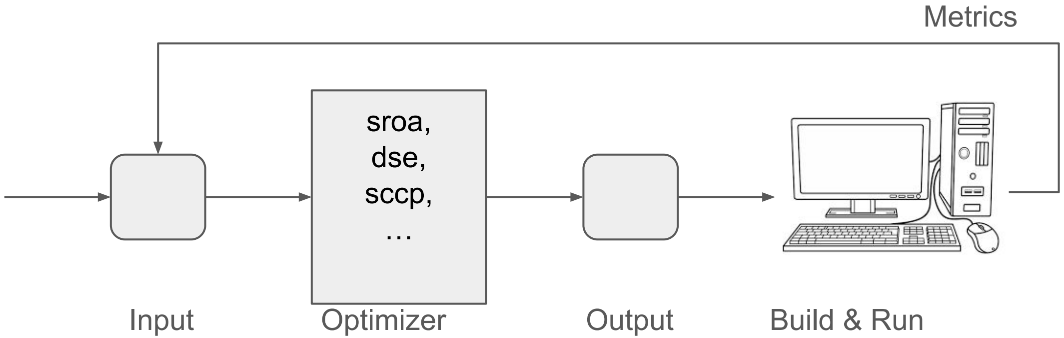
- **Optimal sequences search**
  - Optimization space exploration
  - Benchmarks preparation & analysis
  - Algorithm choosing
    - EA, RL
    - Supervised learning, collaborative filtering

- **Code characterization**
  - By explicit (expert) features-selection
  - Graph-structured representation analysis (flow-aware)
  - By ML

- **Specialization for target platform**
  - Parameterized passes, parameters choosing
  - Prior knowledge about hardware features

Iterative compilation [1]

Sequential code re-compilation with different pass sequences for choosing the best (k-best)



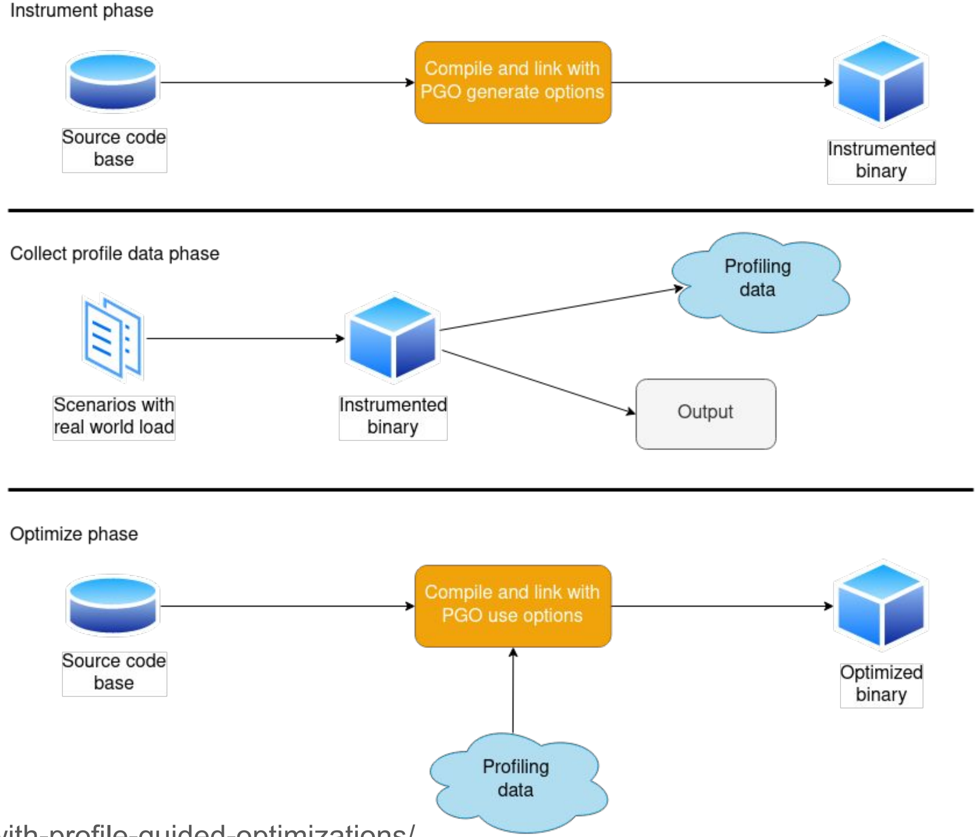
+ Guaranteed best results via brute-force
+ Very simple approach

- Monstrous overhead / time consumption
- No knowledge transfer between experiments

1. Bodin, François & Kisuki, Toru & Knijnenburg, Peter & Boyle, Mike & Rohou, Erven. (2000). Iterative compilation in a non-linear optimisation space. Workshop on Profile and Feedback-Directed Compilation.

Adaptive compilation (“Profiling guided optimization”) [2]

Profiling of instrumented program and applying of the best passes for collected profile



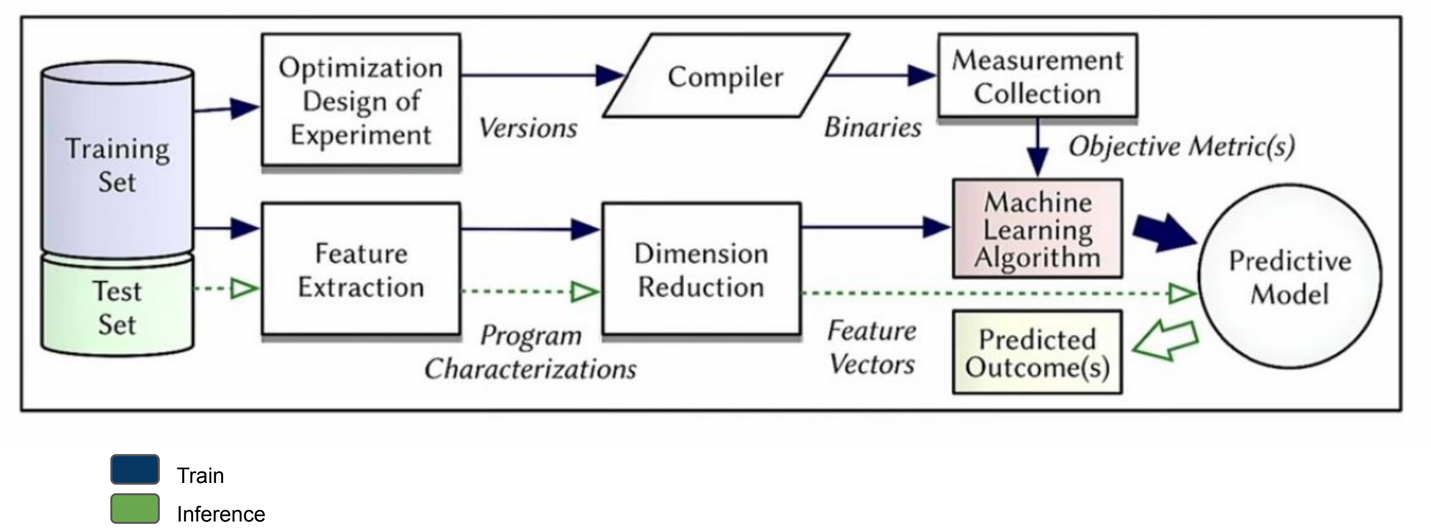
+ Relatively simple approach
+ Approbated in industry (Firefox Mozilla, etc)

- No knowledge transfer
- Best results for certain profiles (and may be worse for another scenarios)

2. https://johnnysswlab.com/tune-your-programs-speed-with-profile-guided-optimizations/

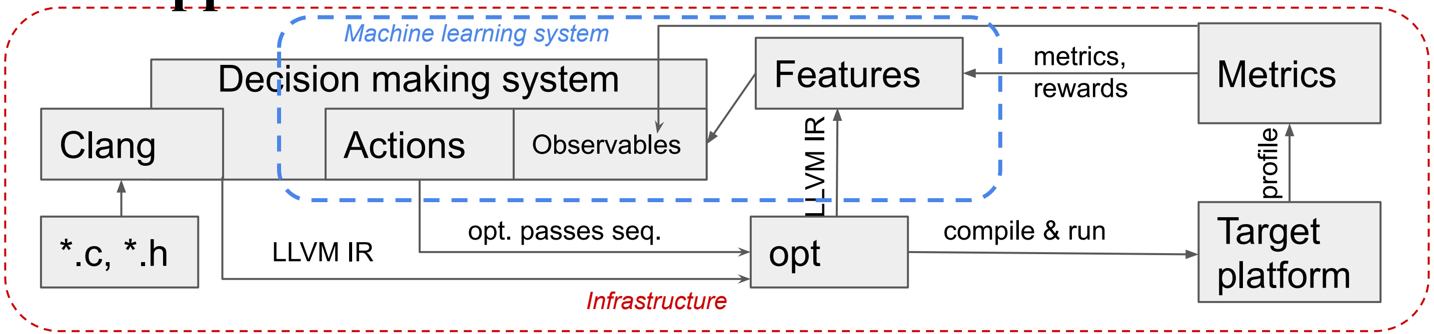
Modern approaches with ML

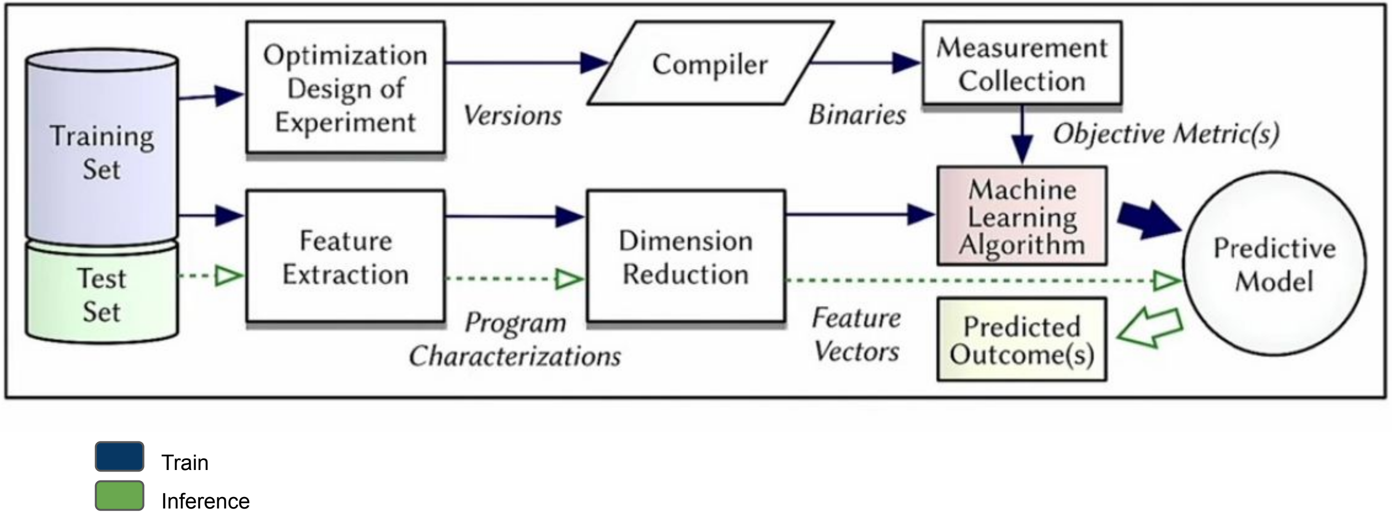
- \+ Knowledge transfer
- \+ Generalization
- \- Quite complex
- \- Require novel code characterization methods, benchmarks preparation



Ashouri, Amir H. & Killian, William & Cavazos, John & Palermo, Gianluca & Silvano, Cristina. (2018). A Survey on Compiler Autotuning using Machine Learning. ACM Computing Surveys. 51. 96:1-. 10.1145/3197978.

Modern approaches with ML





Ashouri, Amir H. & Killian, William & Cavazos, John & Palermo, Gianluca & Silvano, Cristina. (2018). A Survey on Compiler Autotuning using Machine Learning. ACM Computing Surveys. 51. 96:1-. 10.1145/3197978.

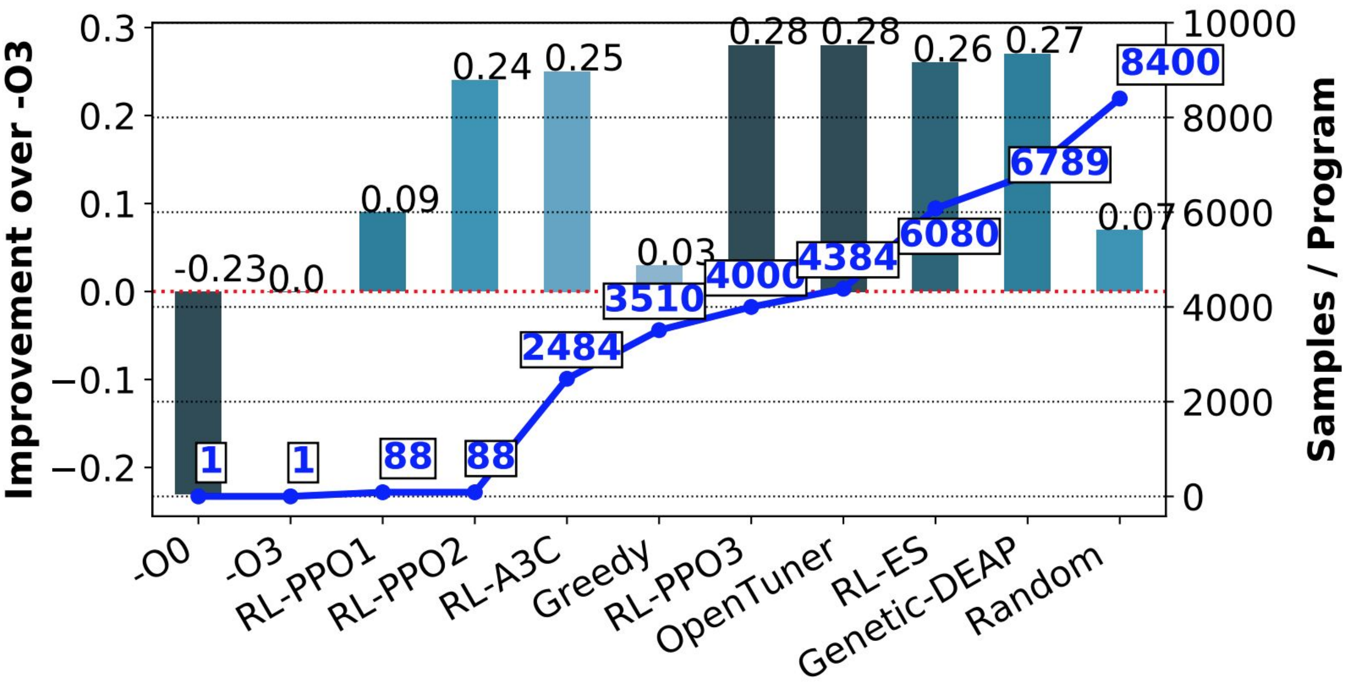
Ускорение поиска и трансфера знаний

Итеративная компиляция заложила основу автоматизированного подхода [Bodin и др., 1998-2000] и определила дальнейший вектор его развития. За ~25 лет успехи следующие:

- Последние десятилетия ускорение поиска осуществлялось в основном за счёт подходов с ГА и стохастический поиск [Tagtekin и др., 2021], а также комбинированием ГА с обучением с учителем [Agakov et al, 2006], etc.
- В последние 5-10 лет начали применять обучение с подкреплением, практические результаты демонстрируют существенно более быструю сходимость к результатам, как у ГА. При этом возникает возможность трансфера знаний
- Задан тренд на построение сложных методов характеризации, в т.ч. посредством обучения представлений
- Разработаны решения, использующие информацию о hardware для лучшей оптимизации (NeuroVectorizer)
- Исследовательские работы, как правило, не связываются со сложными критериями оптимизации (однокритериальная с ограничениями, многокритериальная). В основном, это победа над -O3 / -Os /-Oz Наша группа, наоборот, использует нецелевые характеристики программ как ограничения
- Решения в данной технической области крайне быстро устаревают

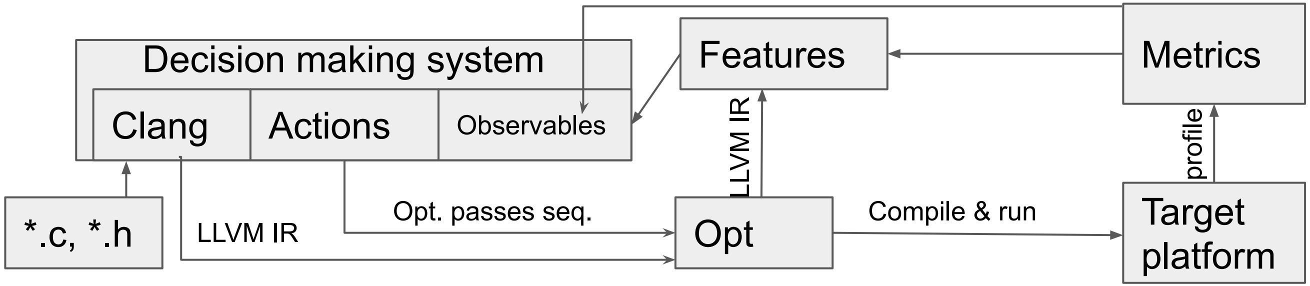
Bodin, François & Kisuki, Toru & Knijnenburg, Peter & Boyle, Mike & Rohou, Erven. (2000). Iterative compilation in a non-linear optimisation space. Workshop on Profile and Feedback-Directed Compilation.
Tağtekin, B., Höke, B., Sezer, M. K., & Öztürk, M. U. (2021, August). FOGA: Flag Optimization with Genetic Algorithm. In 2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA) (pp. 1-6). IEEE.
F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in International Symposium on Code Generation and Optimization (CGO'06). IEEE, 2006, pp. 11–pp.

Appendix #1



https://arxiv.org/pdf/2003.00671.pdf

Appendix #2: Proof-of-concept with heuristic search, LLVM [Efanov, 2023]



Results for CBench .TEXT size reduction without runtime degradation:

benchmark	baseline	Best result	Gain, %
patricia			23.356009070294785
gsm			21.191308866697568
blowfish			10.062516626762436
bzip2			7.324845763802766
jpeg-d			7.229062428095125
Jpeg-c			7.150153217568948
sha			6.361275964391691
tiffmedian			5.311668039060172
tiffdither			5.22980749841004
tiff2bw			5.21401899703284
tiff2rgba			5.20897960871903
qsort			4.854968113556881
stringsearch			3.0260047281323876
stringsearch2			2.7645788336933044
crc32			1.4736842105263157
dijkstra			0.7469654528478058

Method: heuristic search (least from positives per-step);
Iterations num.:100 Iterations;
Episode len.: 15;
Patience: 5;
Runtime eval: 10 times, mean;
Env.: LLVM;
Benchmark: CBench-v1;
Actions: Oz_extra.

Challenges:

- Off-line full sequence prediction (currently solved, can be improved)
- Static code characterization (flow-aware & scalable) methods construction & integration
- Convergence speed-up by subsequences extracted from ODG
- Optimal parameters prediction for parameterized passes (loop unroll, vectorize, inline, etc)
- ML methods improvement (achieved ~11% max size reduction on CBench with AC RL)

Place for slides from En&T-2023 Conference…

Code characterization

Code characterization

- **Static**
  - Aggregated features (for ex, extracted by NLP [3] models or collected from IR by values of instructions, BBs, functions, phi-nodes counters, etc [1,4-5].)
  - Graph-structured (CFG, DFG, CallGraph, etc [4])
  - Mixed

- **Dynamic**
  - Arch-dependent (perf counters, etc) [2-4]
  - Arch-independent (for ex, number of function calls)

- **Hybrid**
  - Collect as much as …

1. Q. Huang, et al., "AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning," in 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 2019 pp. 308-308.

2. POSET-RL: https://arxiv.org/abs/2208.04238

3. [Wang, H, Tang, Z, Zhang, C et al]. (4 more authors) (2022) Automating Reinforcement Learning Architecture Design for Code Optimization. In: CC 2022: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. The 31st ACM SIGPLAN International Conference on Compiler Construction (CC ’22), 02-03 Apr 2022.

4. 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 (December 2020), 27 pages.

5. Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: robust, performant compiler optimization environments for AI research. In Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '22). IEEE Press, 92–105.

Code characterization

- **Static**
  - Aggregated features (for ex, extracted by NLP [3] models or collected from IR by values of instructions, BBs, functions, phi-nodes counters, etc [1,4-5].)
  - Graph-structured (CFG, DFG, CallGraph, etc [4])
  - Mixed

- **Dynamic**
  - Arch-dependent (perf counters, etc) [2-4]
  - Arch-independent (for ex, number of function calls)

Note:

- Несмотря на “раскрученность”, модели на основе NLP (code2vec, CodeBERT, etc) применяются не так часто, что обусловлено многими факторами: нечувствительностью к потоковой информации, трудностью определения семантики (и, как результат, невозможностью обеспечения некоторых свойств представления). Модели, для которых операционная семантика определена, оперируют более сложными структурами контекста, чем скип-грамма из последовательности слов. (“Код -- это не текст”)
- Решения на основе LLM появляются (первая работа -- сентябрь 2023 г.). Качество результатов -- пока что не высокое (3% для уменьшения количества инструкций без ограничений на LLVM). https://arxiv.org/pdf/2309.07062.pdf

Code characterization: from experience view

- NLP-based models are inaccurate (program is not a token sequence, relations between entities should be extracted) [1]. Relations and semantics matters [4,11].
  - Thus, graph-based characterization methods required (or another kind, which take into account the relations on instructions, variables, arguments, types, etc)
  - Theoretically, inst2vec should give the best results in comparison with another, Because of taking into account graph-structure of program flows. But it is not enough.

- Manually-crafted features (intsr.number, bb sizes, trip-counts) are specialization (suitable for certain applications rather than for general purpose)
  - Question of automation is opened -- defaultly, need experts

- Dimensionality reduction required
  - In case IR2Vec, the inference sometimes taken in minutes for relative big programs (for ~600 lines in TU)
  - Sparsity reduction of feature space will lead, at least, to convergence speed gain
- Models
  - Should keep of semantics (demonstrated [11], that CFG,DFG,Call,Type graph-based methods are relatively good in this criteria).
  - Should be extensible to represent static & dynamic features
- The first model with LLMs [https://arxiv.org/pdf/2309.07062.pdf] shows some minor results for unconstrained size reduction.

Code characterization: AutoPhase repr. Vs InstCount

AutoPhase

TotalInsts	567
TotalMemInst	340
testUnary	244
const32Bit	133
NumLoadInst	127
NumEdges	95
NumStoreInst	91
TotalBlocks	73
BBNoPhi	73
NumCallInst	66
NumBrInst	66
BranchCount	66
BlockLow	66
const64Bit	65
numConstOnes	59
NumBitCastInst	56
numConstZeroes	52
onePred	45
oneSuccessor	40

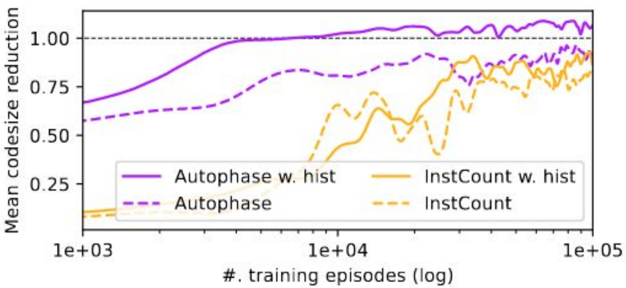
...

InstrCount

TotalInstsCount	567
LoadCount	127
StoreCount	91
TotalBlocksCount	73
CallCount	66
BrCount	66
BitCastCount	56
AllocaCount	37
ICmpCount	29
GetElementPtrCount	19
TotalFuncsCount	14
SExtCount	13
AddCount	13
AndCount	9
TruncCount	8
AShrCount	7
SubCount	6

Accurate choosing of representation leads to:

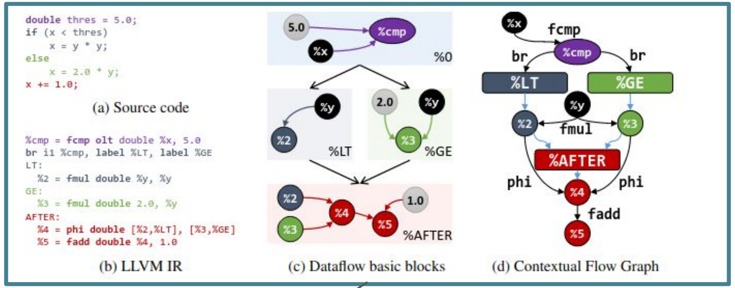
- Convergence speedup
- Results improvement

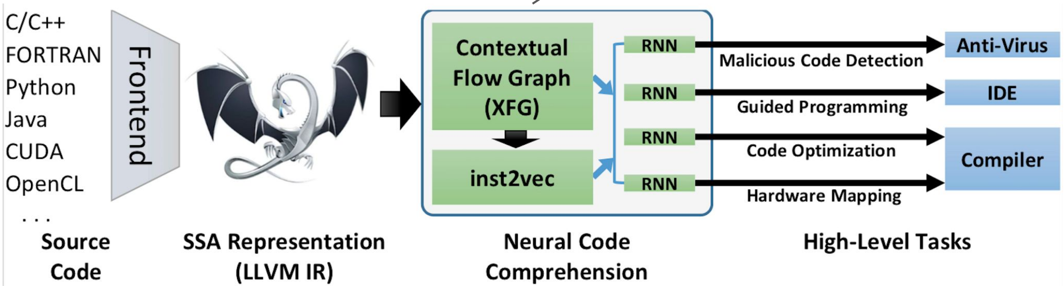


In both cases stronger performance is achieved when coupling the program representation with a histogram of the agent's previous actions. The Autophase representation encodes more attributes of the structure of programs than InstCount and achieves greater performance

Характеризация кода: Inst2vec (2018)

“Statements that occur in the same contexts tend to have similar semantics.”






```
%struct.X = type { i32 }
%3 = getelementptr inbounds %struct.X,
    %struct.X* %0, i64 0, i32 0
```

(a) IR Statement

```
%id = getelementptr inbounds {i32},
    {i32}* %id, i64 %int, i32 %int
```

(b) inst2vec-normalized

Similarity To define similarity, one first needs to define the *semantics* of a statement. We draw the definition of semantics from Operational Semantics in programming language theory, which refers to the effects (e.g., preconditions, postconditions) of each computational step in a given program. In this paper, we specifically assume that each statement modifies the system state in a certain way (e.g., adds two numbers) and consumes resources (e.g., uses registers and floating-point units). It follows that semantic similarity can be defined by two statements consuming the same resources or modifying the system state in a similar way. Using this definition, two versions of the same algorithm with different variable types would be synonymous.

Table 2: Analogy and test scores for `inst2vec`

Context type	Context Size	Syntactic Analogies		Semantic Analogies		Semantic Distance Test
		Types	Options	Conversions	Data Structures	
CFG	1	0 (0 %)	1 (1.89 %)	1 (0.07 %)	0 (0 %)	51.59 %
	2	1 (0.18 %)	1 (1.89 %)	0 (0 %)	0 (0 %)	50.47 %
	3	0 (0 %)	1 (1.89 %)	4 (0.27 %)	0 (0 %)	53.79 %
DFG	1	53 (9.46 %)	12 (22.64 %)	2 (0.13 %)	4 (50.00 %)	56.79 %
	2	71 (12.68 %)	12 (22.64 %)	12 (0.80 %)	3 (37.50 %)	57.44 %
	3	67 (22.32 %)	18 (33.96 %)	40 (2.65 %)	4 (50.00 %)	60.38 %
XFG	1	101 (18.04 %)	13 (24.53 %)	100 (6.63 %)	3 (37.50 %)	60.98 %
	2	**226 (40.36 %)**	**45 (84.91 %)**	**134 (8.89 %)**	**7 (87.50 %)**	**79.12 %**
	3	125 (22.32 %)	24 (45.28 %)	48 (3.18 %)	**7 (87.50 %)**	62.56 %

Table 3: Algorithm classification test accuracy

Metric	Surface Features [49] (RBF SVM + Bag-of-Trees)	RNN [49]	TBCNN [49]	inst2vec
Test Accuracy [%]	88.2	84.8	94.0	**94.83**

1. Read LLVM IR statements once, storing function names and return statements.
2. Second pass over the statements, adding nodes and edges according to the following rule-set:
 (a) Data dependencies within a basic block are connected. Note: that’s way data-flow is taken into account
 (b) Inter-block dependencies (e.g., φ-expressions) are both connected directly and through the label identifier (statement-less edges).
 (c) Identifiers without a dataflow parent are connected to their root (label or program root).
 (d) Calls to external code (e.g., libraries, frameworks) are divided into two categories: statically-(connections) and dynamically-linked (stubs).

3. Achieved XFG becomes a context for inst2vec statement (упрощённый LLVM IR). Then, skip-gram learning is used to learn the embedding (XFG as set of paths)

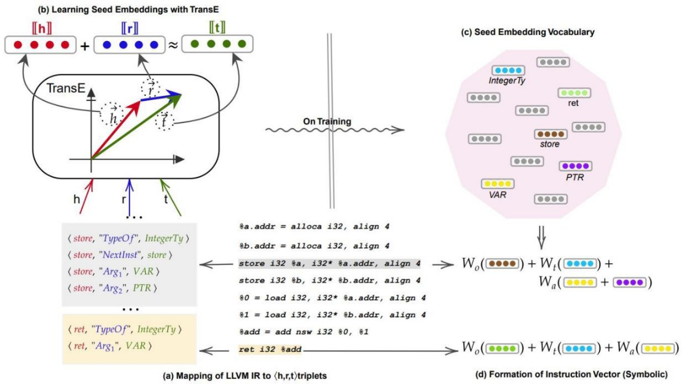
https://github.com/spcl/ncc

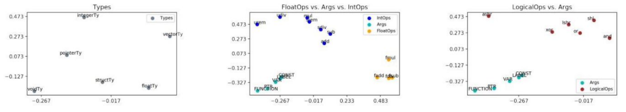
Code characterization: IR2Vec (2020)

Seed dictionary (pre-trained) for LLVM IR entities + combination rules

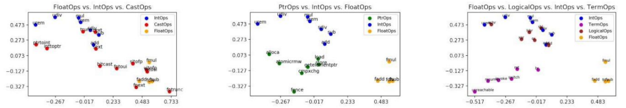
Lets consider triplets $< h, r, t >$, where entities h, t have relation r of one of three types:

1. $TypeOf$ – relation between instruction code and current instruction type
2. $NextInst$ – relation between current and next instruction code
3. Arg_i – between instruction code and its $i - th$ operand





(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

https://arxiv.org/abs/1909.06228

Lets consider the entities, which define the instruction I: code $O^{(I)}$, $i - th$ argument $A_i^{(I)}$, $i \in [0, n]$, type $T^{(I)}$ with vector representations $[O^{(I)}], [A_i^{(I)}], [T^{(I)}]$. Then, the instruction $< O^{(I)}, T^{(I)}, A_0^{(I)}, .., A_n^{(I)} >$ can be represented as vector :

$$W_o[O^{(I)}] + W_t[T^{(I)}] + W_a(\sum_{i=0}^{n}[A_i^{(I)}]), where \tag{1}$$

$W_o > W_t > W_a$ – weight coefficients, learned by TransE method.

Let $RD_0, .., RD_m$ are reaching-definitions of some argument $A_j^{(I)}$ and its vector representations are $[RD_0], .., [RD_m]$. Then, $A_j^{(I)}$ can be represented as:

$$[A_j^{(I)}] = \sum^{m}[RD_i^{(I)}] \tag{2}$$

Note: that's way data-flow is taken into account

Next, for each basic block BB_j representation can be calculated as sum of live instructions $LI_0, .., LI_k$ representations:

$$[BB_j] = \sum_{i=0}^{k}[LI_i] \tag{3}$$

Finally, function F can be represented as sum of its basic blocks $BB_0, .., BB_b$ representations:

$$[F_j] = \sum^{b}[BB_i] \tag{4}$$

Note: that's way control-flow is taken into account

In the same way, the vector of whole translation unit P can be calculated as sum of vector representation of functions $F_1, .., F_f$:

$$[P] = \sum_{i=0}^{f}[F_i] \tag{5}$$

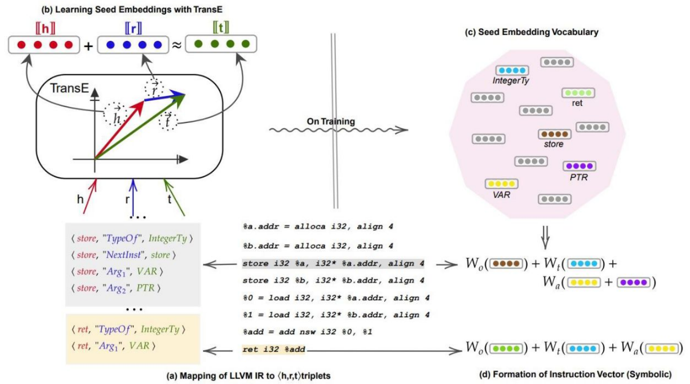
Thus, hierarchical representation method, which can map instructions, types, arguments and relations between them, basic blocks, functions and whole program is constructed.

Code characterization: IR2Vec (2020)

Seed dictionary (pre-trained) for LLVM IR entities + combination rules

Lets consider triplets $< h, r, t >$, where entities h, t have relation r of one of three types:

1. $TypeOf$ – relation between instruction code and current instruction type
2. $NextInst$ – relation between current and next instruction code
3. Arg_i – between instruction code and its $i-th$ operand



Lets consider the entities, which define the instruction I: code $O^{(I)}$, $i-th$ argument $A_i^{(I)}, i \in [0, n]$, type $T^{(I)}$ with vector representations $[O^{(I)}], [A_i^{(I)}], [T^{(I)}]$. Then, the instruction $< O^{(I)}, T^{(I)}, A_0^{(I)}, .., A_n^{(I)} >$ can be represented as vector :

$$W_o[O^{(I)}] + W_t[T^{(I)}] + W_a(\sum_{i=0}^{n}[A_i^{(I)}]), where \qquad (1)$$

$W_o > W_t > W_a$ – weight coefficients, learned by TransE method.

Let $RD_0, .., RD_m$ are reaching-definitions of some argument $A_j^{(I)}$ and its vector representations are $[RD_0], .., [RD_m]$. Then, $A_j^{(I)}$ can be represented as:

$$[A_j^{(I)}] = \sum_{i=0}^{m}[RD_i^{(I)}] \qquad (2)$$

Note: that's way data-flow is taken into account

Next, for each basic block BB_j representation can be calculated as sum of live instructions $LI_0, .., LI_k$ representations:

$$[BB_j] = \sum_{i=0}^{k}[LI_i] \qquad (3)$$

Finally, function F can be represented as sum of its basic blocks $BB_0, .., BB_b$ representations:

Note:

- Data-flow sensitive
- Questions to CFG sensitivity ("+" commutativity)
- Questions to original implementation (NextInst is not used)…
- Previously approved by LLVM-community for MLGO integration
- Quite strange but practically appliable

…am

is constructed.

https://arxiv.org/abs/1909.06228

Code characterization: IR2Vec [4,8] : Experiments & Alternatives

TransE pre-trained LLVM IR entities+manual rules

In [Zavodskikh et al, 2022] Zavodskikh R.K. (my PhD student) have experimentally checked the ability of IR2Vec to save statically estimated reuse-distance between accessed array elements for chosen types of loops. It is determined that the method reflects the instrumentation data in representation vectors, and results of estimation are correlated with Linux Perf measurements. Thus, it demonstrates the ability of IR2Vec to represent control-flow and data-flow accurately enough.

Nevertheless, there are a list of drawbacks of the method detected:

-- BBs are not ordered according to the original CFG order.

-- Original implementation of the method not takes into account NextInst relations (unlike in the method description [4].

Thus, the implementation should be improved.

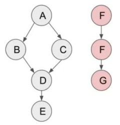
Also, there are alternative methods of flow-aware program features extraction recently published (https://ieeexplore.ieee.org/document/9275317, https://chriscummins.cc/2017/deep-learning-in-compilers), mostly based on graph-structures deep learning via message-passing graph neural networks. But this solutions seems more hard to implement (except ProgramML [11], which already integrated to CompilerGym), re-train and use, in contrast with IR2Vec, in which the basic relations on IR entities are extracted manually and pre-trained as a seeds for representation of compound entities as superpositions of the constituents vectors.

Code characterization: ProGraML (2021)

Graph-structured, flow-aware, type-aware representation, forwarded to MP-GNN

It isn't a vector of numbers

Feature vectors are easy to fool (e.g. insert dead code).



It isn't a sequence of tokens

Sequential representations fail on non-linear relations, long-range deps.

```
void A(int a)
  int b = init();
  //
  // ... 1000 lines
  //
  //
  return b - a;
}
```

Learning with ProGraML: GGNNs

Message Passing

$$M(h_w^{t-1}, e_{wv}) = W_{\text{type}(e_{wv})}\left(h_w^{t-1} \odot p(e_{wv})\right) + b_{\text{type}(e_{wv})}$$

6 typed weight matrices for {forwards,backwards} {control,data,call} edge types

Position gating to differentiate control branches and operand order

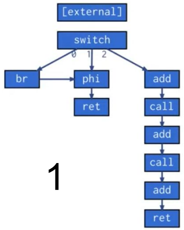
Readout Head

$$R(h_v^T, h_v^0) = \sigma\left(f(h_v^T, h_v^0)\right) \cdot g(h_v^T)$$

per-vertex prediction after T message-passing steps

Embedding construction approach:

- inst2vec for nodes-statements
- Relations learning: MP-GNN
- In some sense, this is mixed approach



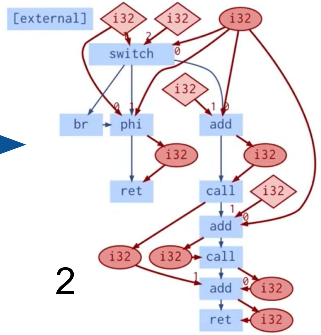
1

Note: Control-Flow

Add graph vertices for **constants** (diamonds) and **variables** (oblongs).

Edges are **data-flow**.

Edge position attribute for **operand order**.



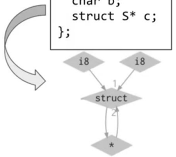
2

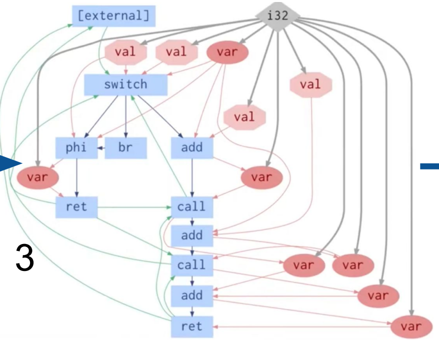
Note: Data-Flow

Nodes represent **types**, Edges are **instances**.

Types are composable. Edge position per field.

```
struct S {
  char a;
  char b;
  struct S* c;
};
```

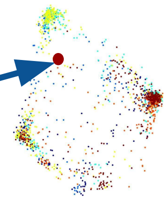




3

Note: CallGraph and TypeInfo*

Learn by Graph NN



Use vertex labels as embedding keys br → 0 add → 1 i32 → 2

Derive vocab from set of unique vertex labels on **training graphs**.

Separate type/instruction nodes leads to **compact vocab**, excellent coverage on unseen programs compared to prior approaches:

	Vocabulary size	Test coverage
inst2vec [12]	8,565	34.0%
CDFG [14]	75	47.5%
ProGraML	2,230	**98.3%** *without types

inst2vec: combined instruction+operands `i32 <id> = a<id> <int8>`

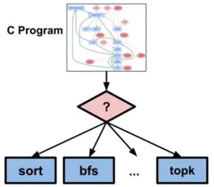
CDFG: uses only instructions for vocab, ignores data

Deep Data Flow

F1 scores

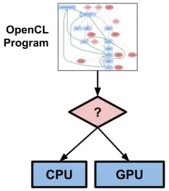
Task	inst2vec	CDFG	ProGraML
Reachability Trivial forwards control-flow E.g. dead code elimination	0.012	**0.998**	0.998
Dominance Forwards control-flow E.g. global code motion	0.004	0.999	1.000
Data Dependencies Forwards data-flow E.g. instruction selection	-	-	0.997
Live-out Variables Backwards control- and data-flow E.g. register allocation	-	-	0.937
Global Common Subexpressions Instruction/operand sensitive E.g. GCS Elimination	0.000	0.009	0.996

1. Algorithm Classification



1.35× improvement over state-of-art

2. Heterogeneous Device Mapping



1.20× improvement over state-of-art

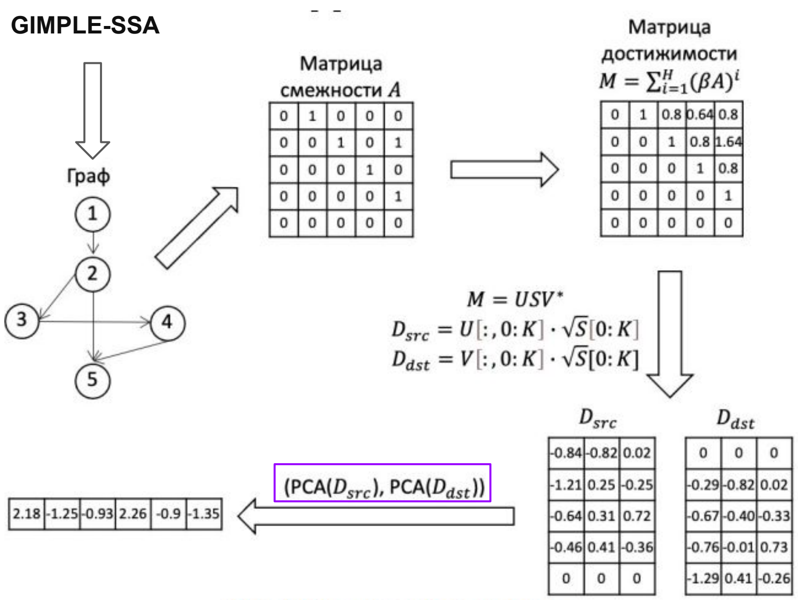
* There some type elision in LLVM IR

ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations (2021) https://chriscummins.cc/pub/2021-icml.pdf

Code characterization: mixed static methods (static features + CFG / VFG)

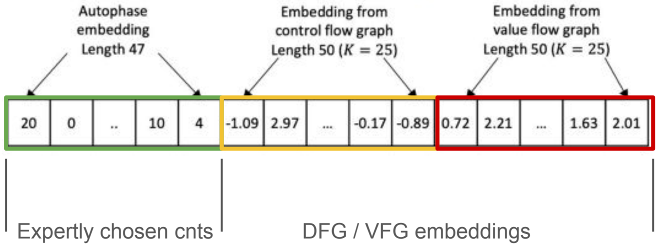
GCC GIMPLE-SSA embedding [Otrashchenko, Akimov, Efanov, 2023]

- GCC IR is characterised using autophase characterisation, control and value flow graphs
- Autophase characterisation consits of information about IR, available immediately during compilation
- The embedding from control flow graph and value flow graph are acquired as shown on the picture



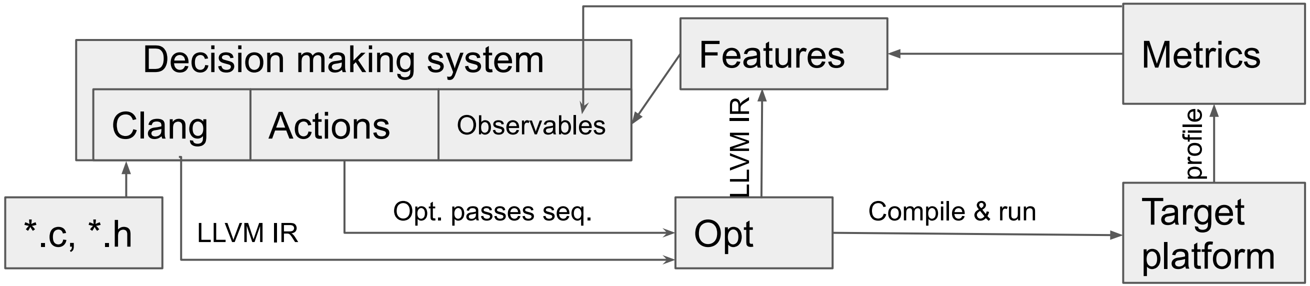
Graph to embedding pipeline

Whole embedding:



RL/GA Approach: GCC PHO (external ppt)

Appendix 1: Proof-of-concept with heuristic rearch



Results for CBench .TEXT size reduction without runtime degradation:

benchmark	baseline	Best result	Gain, %
patricia			23.356009070294785
gsm			21.191308866697568
blowfish			10.062516626762436
bzip2			7.324845763802766
jpeg-d			7.229062428095125
Jpeg-c			7.150153217568948
sha			6.361275964391691
tiffmedian			5.311668039060172
tiffdither			5.22980749841004
tiff2bw			5.21401899703284
tiff2rgba			5.20897960871903
qsort			4.854968113556881
stringsearch			3.0260047281323876
stringsearch2			2.7645788336933044
crc32			1.4736842105263157
dijkstra			0.7469654528478058

Method: heuristic search (least from positives per-step);
Iterations num.:100 Iterations;
Episode len.: 15;
Patience: 5;
Runtime eval: 10 times, mean;
Env.: LLVM;
Benchmark: CBench-v1;
Actions: Oz_extra.

Challenges:

- Off-line full sequence prediction
- Static code characterization (flow-aware & scalable) methods construction & integration
- Convergence speed-up by subsequences extracted from ODG
- Optimal parameters prediction for parameterized passes (loop unroll, vectorize, inline, etc)
- ML methods improvement (achieved ~11% max size reduction on CBench with AC RL)

3. Automating Reinforcement Learning Architecture Design for Code Optimization (2022)

```python
import SuperSonic as ss
from SuperSonic.stateFunctions.models import *
...
statefs = [Word2Vec(...),Doc2Vec(...),CodeBert
      (...),ActionHistory(...)]
tranfs = [DNN(...),CNN(...),LSTM(...)]
rewards = [RelativeMeasure(...),tanh(...)]
rl_algs = [MCTS(...),PPO(...),DQN(...),QLearning
      (...)]
actions = [Init(...)]
...

class SuperOptimizer(ss.PolicyInt):
  def __init__(self, statefs, tranfs, actions,
       rewards, rl_algs):
      self.PolicySpace = {
        "StatList": statefs,
        "TranList": tranfs,
        "ActList": actions,
        "RewList": rewards,
        "AlgList": rl_algs,
      }
      self.search_engine = SearchEngine(self.
           PolicySpace)

  def run(self):
    #user code for compilation and execution
    ...
    return Result(time=run_result['time'])

if __name__=='__main__':
  opt = SuperOptimizer(statefs, tranfs, actions,
       rewards, rl_algs)
  policy = opt.policy_search(
       training_benchmark_list, num_of_trials=100)
```

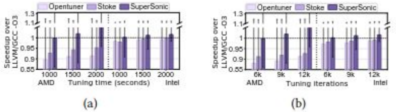


Figure 6. Speedup over LLVM/GCC -O3 for superoptimiza-

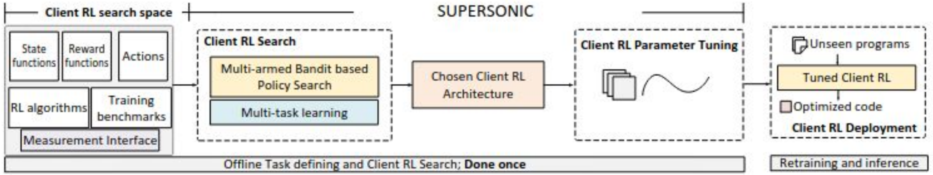


Figure 1. Overview of SuperSonic components. This framework enables developers to express the optimization space. It automatically searches for the optimal client RL architecture to be used for inference during deployment.

Table 2. Case studies in our evaluation

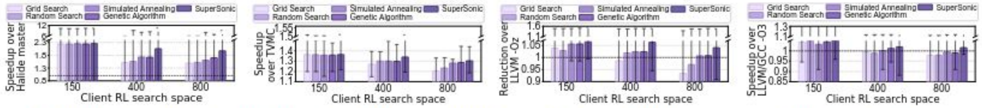
Use cases	#Bench.	Competing Methods	Search space
C1: Optimizating image pipeline	10	Halide master [62], auto-schedulers [3], HalideRL [68], Open-Tuner	$7^3 * 2^7 \sim 7^3 * 3^7$
C2: Neural network code optimization	5	AutoTVM [19], Chameleon[5], Open-Tuner	$5 * 10^3 \sim 20 * 10^3$
C3: Code size reduction	43	CompilerGym [23], OpenTuner	$123^{60} \sim 123^{150}$
C4: Superoptimization	40	STOKE [76], Open-Tuner [9]	$9^{100,000} \sim 9^{16,000,000}$

Table 3. Candidate state functions and reward functions

		Case Studies: 1	2	3	4
State func.	Word2Vec [58]				
	Doc2vec [49]		✓		✓
	CodeBert [29]				
	Manual features (e.g. LLVM IR representation from [40])	✓			
	Action History				
	Hash of Action History			✓	
Reward func.	Relative measure (e.g. speedup, code size reduction ratio)	✓	✓		

Table 5. Search overhead required by SuperSonic to exceed the performance given the best-performing alternative

Use cases	% of search time (& raw numbers)			% of iterations (& raw numbers)		
	MIN	GeoMean	MAX	MIN	GeoMean	MAX
Case study 1	3.6 (21 mins)	39.6 (4 hours)	72.1 (6 hours)	3.9 (1,425)	44.0 (15,821)	75.3 (27,141)
Case study 2	1.0 (1 min)	39.1 (24 mins)	74.1 (45 mins)	4.2 (885)	32.3 (3,885)	68.8 (8,256)
Case study 3	1.5 (3 sec)	31.0 (61 sec)	61.1 (2 mins)	2.1 (738)	29.7 (10,714)	83.5 (30,068)
Case study 4	1.1 (22 sec)	32.8 (11 mins)	42.3 (14 mins)	1.2 (1,478)	36.5 (43,745)	46.9 (56,387)



(a) Optimizing Image Pipelines (b) Neural Network Code Generation (c) Code size reduction (d) Superoptimization

Table 1. Example tunable parameters

Algorithms	Parameters
Common param.	Batch size for workers; Train batch size; Mini-batch size; learning rate
MCTS	Dirichlet noise and epsilon; Puct coefficient; Simulation times; Loss temperature
PPO	Entropy coefficient; Adam optimizer step size; GAE estimator parameter; Policy ratio clipping
DQN	#atoms; Discrete supports; Adam epsilon; Clip gradients
Q-Learning	Behavior Cloning Pretraining numbers; Q-Learning loss temperature; Lagrangian threshold; Min Q weight multiplier

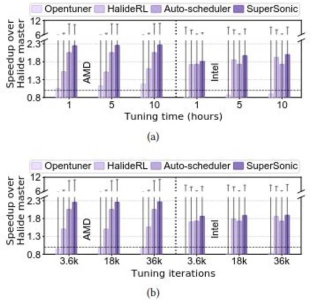


Figure 3. Performance to expert-tuned Halide schedules under different search time (a) and iteration (b) constraints. SuperSonic gives the overall best performance than other auto-tuning methods.

- Code features language models are used -- does not take into account data flow and control flow
- Mainly implements Iterative approaches
- I think that this is some overkill for us

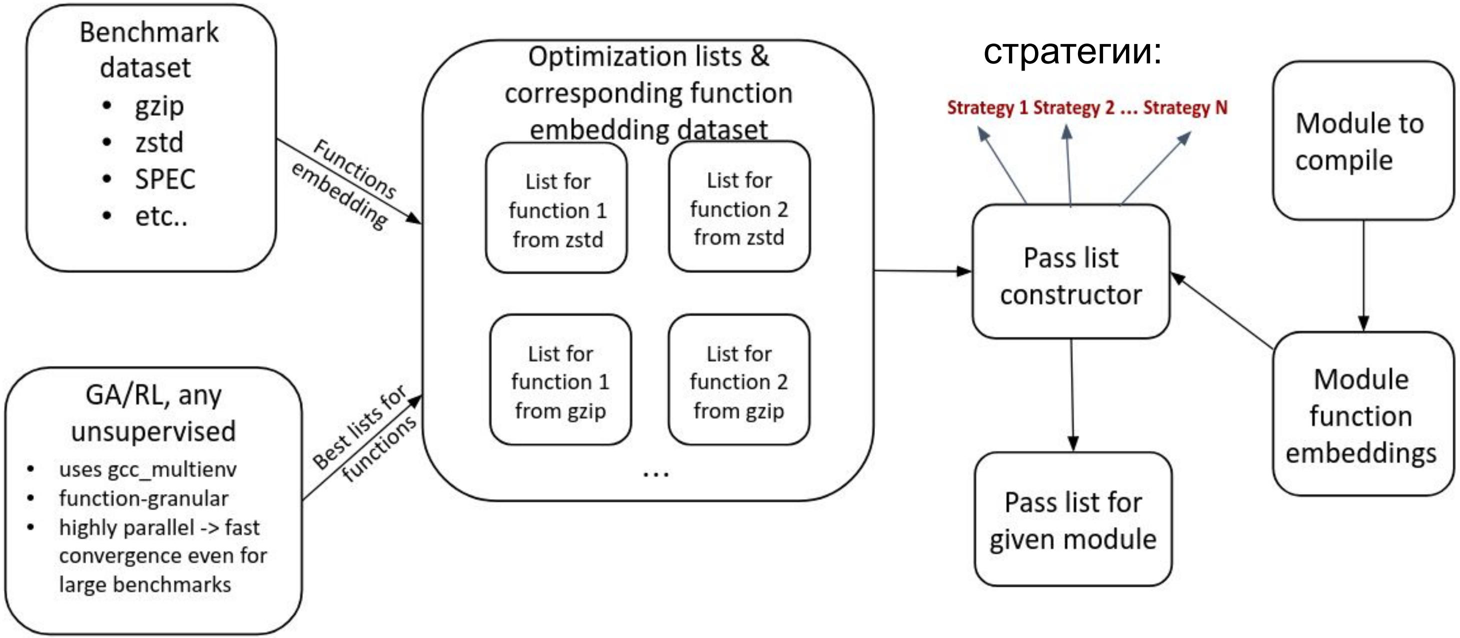
\+ Many different techniques and methods for finding the optimum are integrated, automated decision-making, which technique to use in a particular case

\+ Seems as very powerful and flexible meta-optimization framework for different ML-driven code transformation & optimization tasks

https://zwang4.github.io/publications/cc22.pdf

RL/GA combining with supervised learning

RL / GA + supervised learning

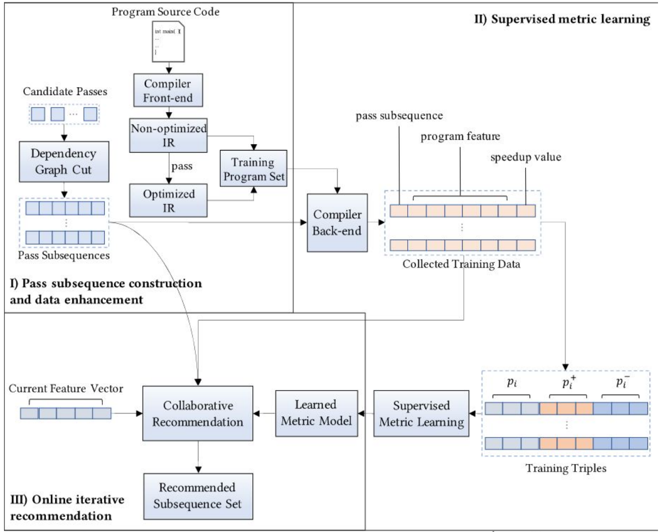


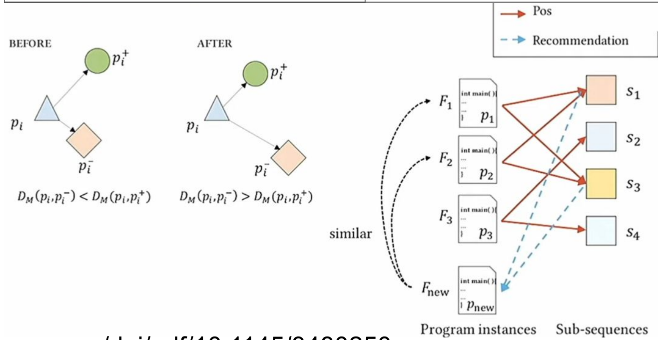
Scenarious:

- **Program classification by classes with known best optimizations**
- **Aggregation of passes to greater granularity ones** (см. Рис.)
    - Best optimizations for functions are known
    - Aggregate the explicit sequence for module compilation (by passing this sequence as options to compiler driver / optimizer):
        - Compilation speed is important (but not critical)
        - Goal is to minimize the loss of quality
        - Mainly, controllability and security (trustworthiness) reasons

Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering (2021) [1]

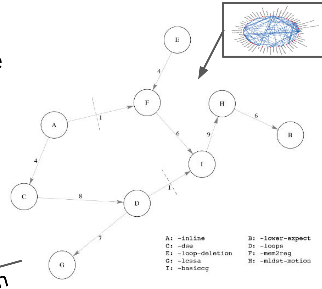
Using the recommendation system approach: learn the metrics between the program representations in that way that programs with similar best pass sequences be closer in embedding space each to another.





https://dl.acm.org/doi/pdf/10.1145/3480250

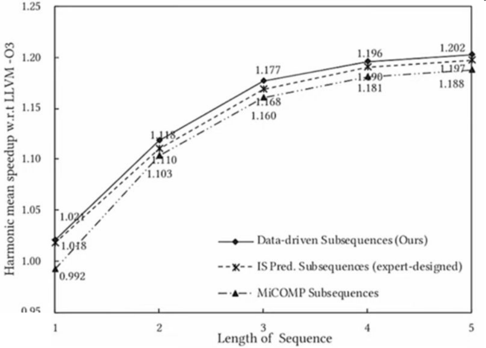
- Consider that pass sequence *(A,B)* is preferable for program *p* than *(A)* and *(B)* separately, and also than *(B,A)*, if.f $f(p,(A,B))>f(p,(A))$; $f(p,(A,B))>f(p,(B))$; $f(p,(A,B))>f(p,(B,A))$, where *f* is the metric of performance gain.
- Such passes, that A improves effectivity of pass *B*, so *B* depends on *A*, are called "collaborative interactions" between *A* and *B*.
- As collaborative interactions are oriented 2-ary relations on passes, the full set of explored pass relations can be represented as oriented dependency graph -- *ODG*.



ODG segmentation

Collaborative filtering:
Recommend on the inference such subsequences that best optimize programs "similar to the current program" in some metric.

[6] : learn metric such that the distance between programs for which the same pass sequences fit is small, and for not same it is large



The article is based on more earlier article [1] with improved Approach for subsequences construction. The [2] based on graph-structured agglomeration, when collaborative filtering with metric learning in [1] gives better results.

[1] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. 2021. Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering. ACM Trans. Arch. Code Optim. 19, 1, Article 2 (December 2021), 25 pages. https://doi.org/10.1145/3480250
[2] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Trans. Archit. Code Optim. 14, 3, Article 29 (September 2017), 28 pages. https://doi.org/10.1145/3124452

Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering (2021) [1]

Using the recommendation system approach: learn the metrics between the program representations in that way that programs with similar best pass sequences be closer in embedding space each to another.

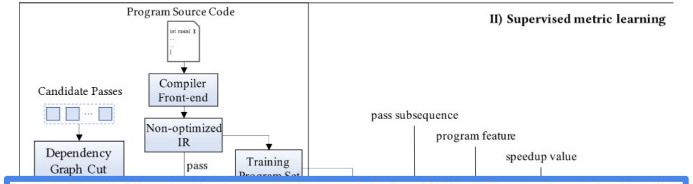


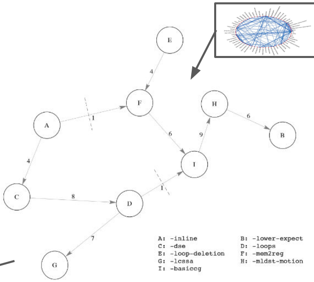
Table 7. Reported Speedup Numbers on the Columns 2 to 4 are A(B%): (A) Speedup (normalized by -O3), and (B) Percentage Speedup w.r.t. Optimal Speedup Value of an Exhaustive Exploration of MiCOMP's RIC. Columns 5, 6, and 7 are the Best Optimization Sub-sequences found using an Exhaustive RIC with MiCOMP's Sub-sequences, Their Corresponding Speedups, and Number of Iterations it Took to Outperform LLVM's -O3 (Total:19k)

Applications	MiCOMP's Prediction			MiCOMP's RIC		
	1 prediction	5 predictions	10 predictions	Best sub-sequence found	Speedup w.r.t -O3	# Iterations to outperform -O3 (K)
automotive_bitcount	1.04 (95.38%)	1.07 (98.12%)	1.08 (98.92%)	BEACCA	1.19	8.53
automotive_qsort1	1.01 (95.32%)	1.03 (96.93%)	1.03 (97.55%)	CBAAAC	1.04	9.41
automotive_susan_c	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)	BDBCCB	1.32	8.1
automotive_susan_e	1.04 (96.47%)	1.03 (98.41%)	1.04 (99.00%)	AABACA	1.15	8.78
automotive_susan_s	0.99 (96.26%)	1.01 (98.42%)	1.02 (98.98%)	ECCCDE	1.22	8.36
bzip2d	0.93 (92.77%)	0.96 (94.02%)	1.00 (94.37%)	CBDACA	1.29	8.24
bzip2e	1.09 (83.77%)	1.10 (86.02%)	1.12 (90.37%)	CBADCA	1.3	8.25
consumer_jpeg_c	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)	DDC	1.41	8.64
consumer_jpeg_d	1.09 (84.70%)	1.14 (88.97%)	1.17 (97.85%)	CCED	1.18	9.74
consumer_ti 2bw	0.96 (75.54%)	0.99 (80.59%)	1.02 (82.46%)	DDCAB	1.15	10.17
consumer_ti 2rgba	0.91 (80.61%)	0.95 (86.19%)	1.07 (88.08%)	DDCA	1.17	10.23
consumer_ti dither	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)	CCDCD	1.3	10.12
consumer_ti median	0.94 (79.21%)	1.02 (85.72%)	1.06 (89.31%)	DEDDC	1.32	10.48
consumer_mad	1.02 (82.14%)	1.09 (85.86%)	1.11 (87.68%)	DCEDCD	1.2	10.34
consumer_lame	0.99 (89.21%)	1.02 (90.72%)	1.06 (92.31%)	BCBACB	1.15	10.51
network_dijkstra	1.13 (60.00%)	1.29 (68.46%)	1.38 (73.00%)	EECBBE	1.51	8.32
network_patricia	0.91 (74.99%)	0.93 (80.79%)	0.97 (93.91%)	CECBAA	1.18	8.55
office_ispell	0.98 (84.99%)	1.01 (90.79%)	1.03 (93.91%)	ABCBAC	1.08	11.22
office_ghostscript	0.99 (79.99%)	1.03 (82.79%)	1.03 (90.91%)	ABEBAE	1.10	10.74
office_rsynth	1.01 (84.99%)	1.02 (90.79%)	1.03 (93.91%)	ABCBA	1.12	10.55
office_stringsearch1	0.98 (64.99%)	1.02 (70.79%)	1.01 (73.91%)	ABCBAC	1.07	10.91
security_sha	0.93 (64.99%)	1.01 (70.79%)	1.03 (73.91%)	DACECA	1.10	12.1
security_blow sh_e	0.97 (64.99%)	1.03 (70.79%)	1.03 (73.91%)	BCCEEA	1.13	12.31
security_blow sh_d	0.97 (64.99%)	0.99 (70.79%)	1.02 (73.91%)	ECEACD	1.10	12.33
security_rijndael_e	0.99 (64.99%)	1.02 (70.79%)	1.01 (73.91%)	CAEEC	1.11	12.12
security_rijndael_d	1.00 (64.99%)	1.01 (70.79%)	1.04 (73.91%)	ACCACE	1.06	12.17
telecom_adpcm_c	0.96 (64.99%)	1.01 (70.79%)	1.02 (73.91%)	ECDDCC	1.35	9.23
telecom_adpcm_d	0.98 (64.99%)	1.02 (70.79%)	1.01 (73.91%)	DCAACA	1.13	10.11
telecom_gsm_d	0.93 (64.99%)	1.03 (70.79%)	1.04 (73.91%)	DCAAC	1.34	9.12
telecom_CRC32	1.01 (85.18%)	1.07 (90.35%)	1.10 (94.51%)	DCAACA	1.26	8.86
telecom_pgp_d	1.04 (96.61%)	1.06 (98.53%)	1.06 (99.07%)	DCAACA	1.21	8.84
telecom_pgp_e	1.02 (80.14%)	1.09 (85.86%)	1.11 (87.68%)	DCA	1.22	9.12
Harmonic mean	**1.01 (84.74%)**	**1.05 (87.51%)**	**1.09 (91.52%)**	-	**1.31**	**9.72**

Program instances Sub-sequences

https://dl.acm.org/doi/pdf/10.1145/3480250

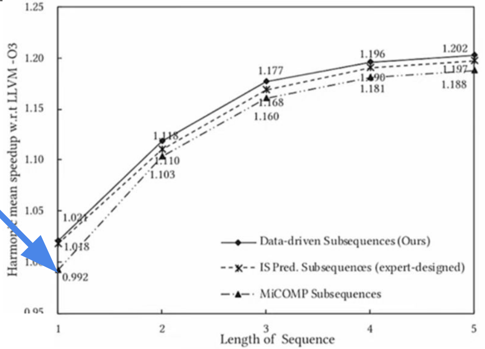
- Consider that pass sequence *(A,B)* is preferable for program *p* than *(A)* and *(B)* separately, and also than *(B,A)*, if.f $f(p,(A,B))>f(p,(A))$; $f(p,(A,B))>f(p,(B))$; $f(p,(A,B))>f(p,(B,A))$, where *f* is the metric of performance gain.
- Such passes, that A improves effectivity of pass *B*, so *B* depends on *A*, are called "collaborative interactions" between *A* and *B*.
- As collaborative interactions are oriented 2-ary relations on passes, the full set of explored pass relations can be represented as oriented dependency graph -- *ODG*.



ODG segmentation

Collaborative filtering:
Recommend on the inference such subsequences that best optimize programs "similar to the current program" in some metric.

[6] : learn metric such that the distance between programs for which the same pass sequences fit is small, and for not same it is large



The article is based on more earlier article [1] with improved Approach for subsequences construction. The [2] based on graph-structured agglomeration, when collaborative filtering with metric learning in [1] gives better results.

[1] Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. 2021. Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering. ACM Trans. Arch. Code Optim. 19, 1, Article 2 (December 2021), 25 pages. https://doi.org/10.1145/3480250
[2] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Trans. Archit. Code Optim. 14, 3, Article 29 (September 2017), 28 pages. https://doi.org/10.1145/3124452

Pass parameters tuning,
Arch-dependent optimizations

Pass parameters tuning / Arch-dependent optimizations

- Mostly unsupervised
- Most simple characterization -- максимально простые
  - Expertly selected features
  - Sometimes code2vec (NLP-based)
- Usually tune a few optimizations (oftenly, only one, but much accurately)
- LLVM-based
- Examples:
  - MLGO (2021)*: https://github.com/google/ml-compiler-opt
  - NeuroVectorizer (2020): https://github.com/intel/neuro-vectorizer

* https://blog.research.google/2022/07/mlgo-machine-learning-framework-for.html

Summary:

- There are a number of not only researched, but also tested in practice solutions for iterative pass order auto-tuning.
- Most solutions look for passes for the entire translation unit
- Multi-step solutions look preferable, in particular, the superiority indicated by the authors of the articles even over expertly selected sequences, while reducing the search time and overhead for iterative measurement. Maybe it deserves consideration ...
- Splitting into subsequences in a multi-step approach is also an open question
- None of the solutions take into account the multi-criteria choice
- None of the phase-ordering/choosing approaches tune the passes parametrization (for ex. ‘UF’ for loop-unroll)
- Most of solutions use a quite inaccurate model for representing (characterizing) programs
  - Ignoring information about data flows
  - Ignoring the control flow
  - Operating:
    - By fixed set of features (for example, the number of instructions of a given type, the size of the BB, etc. - for example, in [1] there are 56 of them, but aggregated as a sum for each of the components of the feature vector)
    - Or by embedding built on mechanisms from the field of natural language processing (word2vec, code2vec, CodeBERT, etc)
    - LLMs are quite new to use it

Review conclusion:

- The most proven solution in practice for now is AutoPhase [1].
  - The used model of the program features [1] is primitive -- IR2Vec (or alternative, for ex. ProgramML[11]) required [4], taking into account data and control flows.
    - It is important for convergency speed-up
  - Improvements are also required in multi-step mode -- selection of subsequences from ODG, and different policies for its implementation [2,3,6].

- None of the solutions take into account the multi-criteria choice, only few solutions are constrained. Some scenarios requires combination with supervised learning
- None of the solutions takes into account the parametrization of passes (Except MLGO and NeuroVectorizers, which focused only on few passes)

- Additionally, the issue of optimizing individual functions (partial compilation) can be considered

Related work

1. Q. Huang, et al., "AutoPhase: Compiler Phase-Ordering for HLS with Deep Reinforcement Learning," in 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 2019 pp. 308-308.

2. POSET-RL: https://arxiv.org/abs/2208.04238

3. [Wang, H, Tang, Z, Zhang, C et al]. (4 more authors) (2022) Automating Reinforcement Learning Architecture Design for Code Optimization. In: CC 2022: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. The 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), 02-03 Apr 2022.

4. 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 (December 2020), 27 pages.

5. Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: robust, performant compiler optimization environments for AI research. In Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '22). IEEE Press, 92–105.

6. Liu et al. (2021) Hongzhi Liu, Jie Luo, Ying Li, and Zhonghai Wu. 2021. Iterative Compilation Optimization Based on Metric Learning and Collaborative Filtering. ACM Trans. Arch. Code Optim. 19, 1, Article 2 (December 2021), 25 pages. https://doi.org/10.1145/3480250

7. Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Trans. Archit. Code Optim. 14, 3, Article 29 (September 2017), 28 pages. https://doi.org/10.1145/3124452

8. Zavodskikh R. K., Efanov N. N., Tomashev D. D. (2022). Using the LLVM Framework for static performance prediction with embedding of intermediate representation. Proceedings of Moscow Institute of Physics and Technology. Vol. 14, 3 (55) pp 34-45.

9. Tağtekin, B., Höke, B., Sezer, M. K., & Öztürk, M. U. (2021, August). FOGA: Flag Optimization with Genetic Algorithm. In *2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)* (pp. 1-6). IEEE.

10. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams, "Using machine learning to focus iterative optimization," in International Symposium on Code Generation and Optimization (CGO'06). IEEE, 2006, pp. 11–pp.

11. PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations (2021) https://chriscummins.cc/pub/2021-icml.pdf