Journal of Computer Science

Original Research Paper

Understanding the Code Transformation Algorithms' Impact

Anderson Faustino da Silva and Leonardo Deganello de Souza

Department of Informatic, State University of Maringá, Maringá, Paraná, Brazil

Article history
Received: 27-02-2019
Revised: 09-08-2019
Accepted: 26-11-2019

Corresponding Author:
Anderson Faustino da Silva
Department of Informatic, State University of Maringa,
Maringa, Parana, Brazil
Email: anderson@din.uem.br

Abstract: Modern compilers provide several code transformations, which are automatic program transformations applied with the goal of improving the program performance. In this article, we investigate how standard compiler code transformations, performed at the compiler intermediate representation, affect such representation and consequently the performance. Our research targets clang/LLVM, a popular compiler infrastructure. Our experimental evaluation demonstrates how several code transformations change the intermediate representation and consequently improve the target code's performance in terms of runtime.

Keywords: Code Transformation, Intermediate Representation, Clang/LLVM

Introduction

The process of code generation, performed by compilers, is divided into two major phases (Aho *et al*., 2006; Sebesta, 2009; Scott, 2009). The first, analysis, analyzes the source code to verify its correctness according to the rules defined by the programming language. The second, synthesis, transforms the source code in the target code. In general, each phase can be subdivided into several sub-phases, where the number of phases will depend on the compiler.

Although the number of phases of modern compilers is different, they have in common a code optimization phase (Muchnick, 1997). This phase aims to change the structure of the source code to improve the target code's performance; however without changing the semantics.

Modern compilers have several Transformation Algorithms (TA), which are well-known as optimizations; although this does not mean that any TA will improve the quality of the target code. Loss of performance may occur because the characteristics of the source code do not fit the characteristics by which a TA was proposed. For example, constant folding, whose objective is to evaluate continuous expressions in compilation time, will be useful when the code provides such expressions.

It is important to note that compilers are designed to be generic. In other words, compilers are designed to generate efficient target code for any source code. Therefore, compilers should be prepared to improve the quality of any source code. In this context, it is essential to which to apply, during the compilation process, will be efficient.

Of course, a question could arise. Why do I need to choose TAs instead of applying all TAs available? It is necessary to choose TAs for two reasons. First, as mentioned previously, the characteristics of the source code may not correspond to the characteristics expected by the TA. Second, although it is expected that a TA improve the quality of the source code the reverse may occur. An example is the use of inline, which replaces a function call by your body. Such transformation can lead to code explosion, which in turn can lead to misuse of the cache; consequently degrading the target code's performance.

In this context, this article aims to analyze the behavior of several TAs applied by the compiler clang/LLVM, at the Intermediate Representation (IR) level and thus understand the relationship between TAs, IR and target code's performance.

The contributions of this article are as follows:

1. We present a detailed analysis of how TAs modify the IR
2. We present the relationship between TAs and target code's performance

Our experimental evaluation demonstrates that although several TAs change the IR, a few are effective in improving the target code's performance in terms of reducing its runtime. In addition, the evaluation demonstrates that a reduction in runtime is directly proportional to the reduction in the number of memory instructions.

The rest of this article is organized as follows. Section 2 presents related work in the context of code transformation algorithms' impact. Section 3 provides some hints on code transformation algorithms. Section 4 describes the experimental setup and outlines the methodology used in all experiments. The next three sections, Section 5, 6 and

7, presents our findings. Section 8 remarks about the presented work and future work.

Related Work

Seng and Tullsen (2003) examine the effect of TA son energy usage and power consumption, in the context of the Intel Pentium 4 processor. The authors evaluate the compiler optimization levels, besides three optimizations, namely, loop unrolling, loop vectorization and inline. They demonstrate that enabling TAs leads to reduce energy consumption; although such reduction comes from a reduction in program runtime.

Eyerman *et al*. (2008) evaluate the performance impact of TAs on superscalar processors. Similar to Seng and Tullsen (2003), Eyerman *et al*. (2008) also evaluates the compiler optimization levels, besides several individual TAs. The results indicate TAshave different performance impact on in-order versus out-of-order processors.

Ibrahim *et al*. (2009) evaluate the influence of the compiler optimization levels on the energy and power consumption, but in the context of embedded software. They demonstrate high compiler optimization levels reduce the runtime, however, increasing power consumption. In addition, they demonstrate TAs may decrease both memory references and data cache miss rate, increasing IPC, as well. However, such behavior consequently increases power consumption.

Foleiss *et al*. (2011) evaluate the effect of TAson code size, in the context of code generation targeting small footprint and low energy consumption. The goal of this research is to identify which TAs are heavily related, in respect to code size reduction. They demonstrate single TAdoes not lead to interesting results and it is necessary to combine several TAsto generate small code.

Instead of evaluating TAs, Lee *et al.* (2012) propose a compiler optimization strategy to reduce cache power with victim cache. Their proposal improves performance, besides reducing power consumption by minimizing accesses to the L2 cache, miss rate and miss penalty. To achieve this goal, they propose an approach which analyzes an application, perform scheduling and then insert the instrumentation instructions in the code to control victim cache. They demonstrate compiler optimizations can improve performance related to cache access.

Dong *et al*. (2015) study how TAsinfluence traditional symbolic execution. They focus on clang/LLVM's TAs, trying to understant how different TAs influence the performance of symbolic execution across different program classes. They demonstrate that applying some TAs in a pre-defined order leads to a slowdown. As a result, they indicate TAsshould be chosen carefully when performing symbolic execution.

Hariri *et al*. (2016) study how TAsaffect the cost and results of mutation testing performed at the compiler intermediate representation. They demonstrate that using high compiler optimization levels, the total number of mutants generated is higher than at a low compiler optimization level. This means mutation testing can use very high compiler optimization levels.

Yuan *et al*. (2018) evaluate the TAs effect on system-level near field EMI. They demonstrate that different TAs have large EMI impact for the same program.

As can be observed in this section, these researches differ from our research because our goal is to understand the relationship between TAs, IR and target code's performance and not only evaluating the TAs impact on performance.

Code Transformation Algorithms

Modern compilers (Aho *et al.*, 2006) provide several code transformations (Muchnick, 1997), which can be turned on or off during target code generation, to improve the target code quality. However, it is a difficult task to discover what transformations should be turned on or off. To address this issue, compiler developers provide several code transformation sequences, well-known as compiler optimization levels.

A compiler optimization level is composed of analysis and transformation algorithms. The former analyzes the code and adds instrumentation on it. The later transforms the code hoping to improve the performance.

The clang/LLVM compiler provides three levels for runtime performance, as follows:

O1: This level applies transformations which optimize quickly
O2: This level applies transformations practically certain to produce better performance
O3: This level applies transformations likely to have a beneficial effect

Table 1 Presents the analysis and transformations algorithms existing in clang/LLVM compiler, which are applied at compiler IRlevel.

The sequences of the three clang/LLVM optimization levels represent the order of application, of each analysis and/or transformation algorithm.

It is important to realize that some algorithms are repeated. This happens for one out of two reasons, a specific transformation needs a specific analysis or some transformations open space for another.

The clang/LLVM optimization levels comprise 130, 142 and 143 algorithms, respectively for O1, O2 and O3. However, only 64 different algorithms are used, 19 analysis and 45 transformations.

It is possible to realize that O1, O2 and O3 are actually clang/LLVM optimization levels. The level O2 removes always-inline from O1 and adds mldst-motion, constmerge, elim-avail-extern, globaldce, gvn, inline, mldst-motion and slp-vectorizer.

DOI: 10.3844/jcssp.2019.1678.1693

Table 1: Clang/LLVM Analysis/Transformations

O1	targetlibinfo tti tbaa scoped-noalias assumption-cache-tracker forceattrs inferattrs ipsccp globalopt domtree mem2reg deadargelim basicaa aa domtree instcombine simplifycfg basiccg globals-aa prune-eh always-inline functionattrs domtree sroa early-cse lazy-value-info jump-threading correlated-propagation simplifycfg basicaa aa domtree instcombine tailcallelim simplifycfg reassociate domtree loops loop-simplify lcssa loop-rotate basicaa aa licm loop-unswitch simplifycfg basicaa aa domtree instcombine loops scalar-evolution loop-simplify lcssa indvars aa loop-idiom loop- deletion loop-unroll basicaa aa memdep memcpyopt sccp domtree demanded-bits bdce basicaa aa instcombine lazy- value-info jump-threading correlated-propagation domtree basicaa aa memdep dse loops loop-simplify lcssa aa licm adce simplifycfg basicaa aa domtree instcombine barrier basiccg rpo-functionattrs basiccg globals-aa float2int domtree loops loop-simplify lcssa loop-rotate branch-prob block-freq scalar-evolution basicaa aa loop-accesses demanded-bits loop- vectorize instcombine simplifycfg basicaa aa domtree instcombine loops loop-simplify lcssa scalar-evolution loop-unroll basicaa aa instcombine loop-simplify lcssa aa licm scalar-evolution alignment-from-assumptions strip-dead-prototypes verify
O2	targetlibinfo tti tbaa scoped-noalias assumption-cache-tracker forceattrs inferattrs ipsccp globalopt domtree mem2reg deadargelim basicaa aa domtree instcombine simplifycfg basiccg globals-aa prune-eh inline functionattrs domtree sroa early-cse lazy-value-info jump-threading correlated-propagation simplifycfg basicaa aa domtree instcombine tailcallelim simplifycfg reassociate domtree loops loop-simplify lcssa loop-rotate basicaa aa licm loop-unswitch simplifycfg basicaa aa domtree instcombine loops scalar-evolution loop-simplify lcssa indvars aa loop-idiom loop-deletion loop-unroll basicaa aa mldst-motion aa memdep gvn basicaa aa memdep memcpyopt sccp domtree demanded-bits bdce basicaa aa instcombine lazy-value-info jump-threading correlated-propagation domtree basicaa aa memdep dse loops loop-simplify lcssa aa licm adce simplifycfg basicaa aa domtree instcombine barrier basiccg rpo-functionattrs elim-avail-extern basiccg globals-aa float2int domtree loops loop-simplify lcssa loop-rotate branch-prob block-freq scalar-evolution basicaa aa loop-accesses demanded-bits loop-vectorize instcombine scalar-evolution aa slp-vectorizer simplifycfg basicaa aa domtree instcombine loops loop-simplify lcssa scalar-evolution loop-unroll basicaa aa instcombine loop-simplify lcssa aa licm scalar-evolution alignment-from-assumptions strip-dead-prototypes globaldce constmerge verify
O3	targetlibinfo tti tbaa scoped-noalias assumption-cache-tracker forceattrs inferattrs ipsccp globalopt domtree mem2reg deadargelim basicaa aa domtree instcombine simplifycfg basiccg globals-aa prune-eh inline functionattrs argpromotion domtree sroa early-cse lazy-value-info jump-threading correlated-propagation simplifycfg basicaa aa domtree instcombine tailcallelim simplifycfg reassociate domtree loops loop-simplify lcssa loop-rotate basicaa aa licm loop- unswitch simplifycfg basicaa aa domtree instcombine loops scalar-evolution loop-simplify lcssa indvars aa loop-idiom loop-deletion loop-unroll basicaa aa mldst-motion aa memdep gvn basicaa aa memdep memcpyopt sccp domtree demanded-bits bdce basicaa aa instcombine lazy-value-info jump-threading correlated-propagation domtree basicaa aa memdep dse loops loop-simplify lcssa aa licm adce simplifycfg basicaa aa domtree instcombine barrier basiccg rpo-functionattrs elim-avail-extern basiccg globals-aa float2int domtree loops loop-simplify lcssa loop-rotate branch-prob block-freq scalar-evolution basicaa aa loop-accesses demanded-bits loop-vectorize instcombine scalar-evolution aa slp-vectorizer simplifycfg basicaa aa domtree instcombine loops loop-simplify lcssa scalar-evolution loop-unroll basicaa aa instcombine loop-simplify lcssa aa licm scalar-evolution alignment-from-assumptions strip-dead-prototypes globaldce constmerge verify
O1	aa adce alignment-from-assumptions always-inline argpromotion assumption-cache-tracker barrier basicaa basiccg bdce block-freq branch-prob constmerge correlated-propagation deadargelim demanded-bits domtree dse early-cse
U	elim-avail-extern float2int functionattrs globals-aa globaldce globalopt gvn indvars inferattrs inline instcombine ipsccp
O2	jump-threading lazy-value-info lcssa licm loop-accesses loop-deletion loop-idiom loop-rotate loop-simplify loop-unroll
U	loop-unswitch loop-vectorize loops mem2reg memcpyopt memdep mldst-motion prune-eh reassociate rpo-functionattrs
O3	scalar-evolution sccp scoped-noalias simplifycfg slp-vectorizer sroa strip-dead-prototypes tailcallelim targetlibinfo tbaa tti verify
An.	aa basicaa basiccg block-freq branch-prob demanded-bits domtree functionattrs globals-aa inferattrs lazy-value-info loops memdep rpo-functionattrs scoped-noalias targetlibinfo tbaa tti verify
Tr.	adce alignment-from-assumptions always-inline argpromotion assumption-cache-tracker barrier bdce constmerge correlated-propagation deadargelim dse early-cse elim-avail-extern float2int globaldce globalopt gvn indvars inline instcombine ipsccp jump-threading lcssa licm loop-accesses loop-deletion loop-idiom loop-rotate loop-simplify loop-unroll loop-unswitch loop-vectorize mem2reg memcpyopt mldst-motion prune-eh reassociate scalar-evolution sccp simplifycfg slp-vectorizer sroa sroa strip-dead-prototypes tailcallelim
Rep.	aa(13,16,16)[1] basicaa(10,11,11) basiccg(2,2,2) correlated-propagation(1,1,1) demanded-bits(1,1,1) domtree(10,10,10) globals-aa(1,1,1) instcombine(7,7,7) jump-threading(1,1,1) lazy-value-info(1,1,1) lcssa(5,5,5) licm(1,2,2) loop-rotate(1,1,1) loop-simplify(5,5,5) loop-unroll(1,1,1) loops(4,4,4) memdep(1,2,2) scalar-evolution(3,4,4) simplifycfg(5,5,5)

[1]X(a,b,c) = X: analysis/transformation algorithm, a: O1 repetitions, b: O2 repetitions, c: O3:repetitions

In turn, the level O3 adds to the sequence O2 argpromotion. As can be seen in Table 1, the repetition of some algorithms follows the same pattern for the levels O2 and O3. In addition, these levels increase the repetitions of the aa, basicaa, licm, memdep and scalar-evolution algorithms when compared to the level O1.

Details about each TAcan be found at www.llvm.org.

Methodology

To analyze the behavior of TAs, we perform several experiments. In addition to describing how the experiments are performed, this section describes the experimental setup and outlines the methodology used in all experiments.

Hardware and Software

- **Hardware Intel:** Core i7-3770 processor with a frequency of 3.40GHz, 8 MB cache and 8 GB of RAM
- **Operating system:** Debian 4.9.110-3 with kernel 4.9.0-8- amd.
- **Compilation system:** We use clang/LLVM (Lattner and Adve, 2004; Team, 2019).
- **Code transformation:** We evaluate the TAs: [O1UO2 UO3]
- **Benchmarks:** SPEC CPU2006 Benchmark suite (Henning, 2006; Corporation, 2019), with train dataset

Evaluating the TA Impact at the IR

To fulfill the goal of this research, we evaluate the TA impact at the IR using two different features. They are:

1. a CFG-based features and
2. a DNA-based features

The features are collected at compile time, after the compiler applies a TA, by modules we implemented for this purpose.

In our experiments, each program is compiled using a single TA, besides compiling without using a TA. In this way, it is possible to evaluate the transformations performed at the IRlevel.

In addition, we evaluate the TA impact at the IR in terms of performance, collecting the program runtime.

The CFG-Based Features

The Control Flow Graph (CFG) features are a subset of the features proposed by Namolaru *et al*. (2010), which are numeric features extracted from relationships between the program entities. The importance of these features is due to Namolaru *et al*. (2010) prove their influence on parameterizing code-generating systems, to generate good target code.

We use a subset of the aforementioned features because some features are related to instructions' programs. We will evaluate the instructions' programs, however, using different features.

Table 2 shows the CFG-based features extracted from the IRand used during the analysis.

The DNA-based Features

The DNA-based features characterize each instruction of the IRas a gene, which composes a DNA.

DNA has a great advantage as program features, it captures all of the program's structures and encodes all of its instructions simultaneously.

The clang/LLVM IR is composed of 57 instructions. Thus, we have 57 different genes. These genes is presented in Table 3.

The genes can be grouped into eight groups, as follows:

- Terminator instructions: A, B, C, D, E, F, G
- Binary operations: H, I, J, K, L, M, N, O, P, Q, R, S
- Bitwise operations: T, U, V, X, W, Y
- Vector instructions: Z, a, b
- Aggregate instructions: c, d
- Memory instructions: e, f, g, h, i, j, k
- Conversion instructions: l, m, n, o, p, q, r, s, t, u, v, x, w; and
- Other instructions: y, z, 0, 1, 2, 3, 4

Metrics to Evaluate CFG-based Features

Our evaluation uses the following metrics: diversity, decreasing value, increasing value, removal, adding and rate of change.

An important issue to be considered is the diversity of programs. This is an important metric because if there were no diversity, it would be difficult to understand how TAs perform in programs with distant structures. In fact, in the absence of diversity would be necessary to change the set of programs.

Decreasing and increasing value, removal and adding evaluate how a TA changes a feature. As a result, it is possible to understand what modifications are made to the CFG. In addition, we want to evaluate the rate of change in the CFG.

Metrics to Evaluate DNA-based Features

We use a DNA-approach to evaluate the changes made into the IR. For this purpose, we use the algorithm proposed by Needleman and Wunsch (1970).

Needleman and Wunsch propose an optimal global alignment algorithm to find similarities between two biological sequences. The iterative algorithm considers all possible pair combinations that can be constructed from two amino-acid sequences. As a result, the algorithm returns a score which indicates the similarity between two amino-acid sequences.

DOI: 10.3844/jcssp.2019.1678.1693

Table 2: CFG-based features

(F01) phi-nodes in BBs
(F02) edges in the control flow graph
(F03) critical edges in the control flow graph (F04) BBs
(F05) BBs with a single pred (F06) BBs with a single succ
(F07) BBs with a single pred and a single succ (F08) BBs with a single pred and two succs (F09) BBs with two pred
(F10) BBs with two succs
(F11) BBs with two pred and one succ (F12) BBs with two pred and two succs (F13) BBs with more than two preds (F14) BBs with more than two succs (F15) BBs with more than two succs and
more than two preds
(F16) BBs with number of insts greater than 500 (F17) BBs with number of insts in [15, 500] (F18) BBs with number of insts less than 15 (F19) BBs with no phi-nodes
(F20) BBs with less than three phi-nodes (F21) BBs with more than three phi-nodes
(F22) average of instructions per basic block

BBs = basic blocks
pred = predecessor, preds = predecessors
succ = successor, succs = successors
insts = instructions

Table 3: DNA-based features

Instruction	Gene	Instruction	Gene
Br	A	InsertValue	d
Switch	B	Load	e
IndirectBr	C	Store	f
Ret	D	Alloca	g
Invoke	E	Fence	h
Resume	F	AtomicRMW	i
Unreachable	G	AtomicCmpXchg	j
Add	H	GetElementPtr	k
Sub	I	Trunc	l
Mul	J	Zext	m
Udiv	K	Sext	n
Sdiv	L	UIToFP	o
Urem	M	SIToFP	p
Srem	N	PtrToInt	q
Fadd	O	IntToPtr	r
Fsub	P	BitCast	s
Fmul	Q	AddrSpace	t
Fdiv	R	FPTrunc	u
Frem	S	FPExt	v
Shl	T	FPToUI	x
LShr	U	FPToSI	w
Ashr	V	Icmp	y
And	X	FCmp	z
Or	W	Select	0
Xor	Y	VAArg	1
ExtractElement	Z	LandingPad	2
InsertElement	a	PHI	3
ShuffleVector	b	Call	4
ExtractValue	c		

Based on the results provided by this aforementioned algorithm, we use the following metrics: matches, mismatches and indels.

Matches indicate how many genes, from two DNA sequences, are similar. Mismatches indicate the otherwise. Finally, indels indicate how many gaps there are in DNA(s).

In addition, we evaluate the diversity of programs. However, from a different perspective of view.

Metric to Evaluate Performance

We also evaluate the performance of each TA, in terms of the target code's runtime. The performance metric measures the improvement of the target code when the system applies a TA during target code generation, over the target code without applying a TA. Such metric (improvement) is measured as follows:

$$Improvement = \left(\frac{runtime\_with\_TA}{runtime\_without\_TA} * 100 \right) - 1$$

Collecting Data

The results are based on the arithmetic average of five executions. In addition, the machine workload was as minimal as possible.

Evaluating the CFG-Based Features

Diversity

According to CFG-based features, the programs are clustering as shown in Fig. 1.

Two programs stand out, PERLBENCH and GCC. The former is next to a cluster consisting of 3 programs. While the latter stands out among all other programs. Also, if we formed only 2 clusters, one would be formed by GCC while the other programs would form the second cluster.

There is an imbalance between the five clusters, concerning the number of programs that compose them. Only two clusters comprise 70% of the programs, i.e. 12 programs. However, even in the face of such imbalance, there is diversity among such programs.

Our Findings

Tables 4 and 5 indicate the changes made by a TA in each CFG-based feature, the number of programs affect by a TA (#P) and the average rate of change performed in a feature's value (Avg). The changes are indicated as follows:

1. Decreasing value (↓)
2. Increasing value (↑)
3. Removal (-)
4. Adding (+) and
5. No modification (=)

As an example, the entry =↑↓ indicates the feature F01had different behaviors for different programs, when the programs were compiled applying adce. In other words, for the feature F01 three different situations occurred depending on the program.

Tables 4 and 5 shows results for 30 TAs, out of 64 available in [O1∪ O2∪ O3], because only 30 TAscause changes in the CFG-based features.

Based on CFG-based features, TAs can be classified into two groups: (1) those that are weakly influenced by the code structure and (2) those that are heavily influenced by the code structure.

A TA of the first group is one that can be applied indiscriminately in any program, which will cause changes in its CFG. This occurs with 9 TAs: early-cse, gvn, inline, instcombine, jump-threading, loop-rotate, mem2reg, simplifycfg and sroa.

On the other hand, a TA of the second group should be chosen according to the structure of the program. This is necessary due to such TA need specific structures to cause a transformation in the program's CFG. Therefore, 9 TAs (14%) change the CFG-based features of the entire universe of evaluated programs, 21 TAs (33%) change the CFG-based features of a sub-set of evaluated programs, while 34 TAs (53%) do not affect any program evaluated.

Few TAs add or remove features. Only 6 TAs add features: Gvn, inline, jump-threading, mem2reg, sroa and tailcallelim.

The removal of features only occurs in the application of 4 TAs: simplifycfg, early-cse, mem2reg and sroa.

The TAs mem2reg and sroa are special cases. Such TAscan change, increase, or reduce the value of a feature, as well as remove or add features.

Observing the rate of change in the value of a feature is possible grouping the TAs into six distinct groups, they are:

- *rate* > 200%: mem2regand sroa
- 60% < *rate* < 90%: gvn and instcombine
- 10% < *rate* < 40%: jump-threading, loop-rotate and simplifycfg
- 7% < *rate* < 9%: early-cseand inline
- 1% < *rate* < 2%: indvars, licm, loop-deletion, loop-idiom, loop-unroll, loop-unswitch and loop-vectorize; and
- *rate* <1%: adce, bdce, correlated-propagation, dse, globalopt, ipsccp, lcssa
- loop-simplify, memcpyopt,
- prune-eh, reassociate, sccp, slp-vectorizer and tailcallelim

A few TAs change the CFG-based features considerably. Depending on the rate used, we can only consider two or four TAs. In this context, five TAshas a modest effect on the CFG-based features. On the other hand, 47% has a small effect (almost insignificant) on the CFG-based features.

An important issue to be evaluated is how much such changes in the CFG, of a particular program, affect performance. This issue will be assessed after.

The results indicate compiler users should know the structure of your program and the characteristics of a particular TA, to choose which TAs apply to your code. As mentioned before, in an attempt to minimize such probe compiler developers provide various compiler optimizations levels. In this way, the user does not need to choose what TAs to apply, instead, he/she can just use a pre-defined TA sequence. However, several researches in the literature indicate the best approach, in search for performance, is to create a specific sequence for each program (Tartara and Crespi Reghizzi, 2013; De Lima *et al*., 2013a; 2013b; Queiroz Junior and da Silva, 2015; Siraichi *et al*., 2016; Filho *et al*., 2018).

DOI: 10.3844/jcssp.2019.1678.1693

Table 4: Results for CFG-based features (Part I)

	Features												
Transformation	F01	F02	F03	F04	F05	F06	F07	F08	F09	F10	F11	#P	jAvgj
Adce	=↓	=	=	=	=	=	=	=	=	=	=	7	0.81
Bdce	=↓	=	=	=	=	=	=	=	=	=	=	7	0.81
Correlated-propagation	=↓	=	=	=	=	=	=	=	=	=	=	1	0.08
Dse	=↓	=	=	=	=	=	=	=	=	=	=	7	0.04
early-cse	=↓	=	=	=	=	=	=	=	=	=	=	17	7.91
globalopt	=	=	=	=↓	=	=	=	=	=	=	=	2	0.05
gvn	↑	↓	=↓	↓	↓	↓	=	=	=	=	=	17	64.40
indvars	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.07
inline	=↑	↑	=↑↓	↑↓	↑	↑↓	=	=	↑	↑↓	=	17	8.54
instcombine	↑	=	=	=	=	=	=	=	=	=	=	17	88.71
ipsccp	=↓	=↓	=↓	=↓	=↑↓	=↑↓	=	=	=↓	=↓	=	11	0.06
jump-threading	↑↓	↓	↑	↓	↓	↓	=	=	↓	= ↓	=	17	30.42
lcssa	=↑	=	=	=	=	=	=	=	=	=	=	4	0.58
licm	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.09
loop-deletion	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.07
loop-idiom	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.07
loop-rotate	=↑↓	↑	↑	↑	=↑	=↑	=	=	↑	↑	=	17	10.88
loop-simplify	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	0.98
loop-unroll	=↑↓	↑=	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.02
loop-unswitch	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.07l
oop-vectorize	=↑↓	=↑	=↓	=↑	=↑	=↑	=	=	=↑	=	=	16	1.07
mem2reg	↑	=	=	=	=	=	=	=	=	=	=	17	222.72
memcpyopt	=	=	=	=	=	=	=	=	=	=	=	2	0.01
prune-eh	=↓	=↓	=↓	=↓	=↓	=↑	=	=	=↑↓	=↓	=	5	0.64
reassociate	=↓	=	=	=	=	=	=	=	=	=	=	14	0.63
sccp	=↓	=	=	=	=	=	=	=	=	=	=	3	0.01
simplifycfg	=↓-	↓	↑↓	↓	↓	↓	=	=	↓	↓	=	17	21.42
slp-vectorizer	=	=	=	=	=	=	=	=	=	=	=	1	0.03
sroa	↑	=	=	=	=	=	=	=	=	=	=	17	225.13
tailcallelim	=↑	=↑↓	=↓	=↑	=↑	=↑↓	=	=	=↑↓	=	=	8	0.26

Table 5: Results for CFG-based features (Part II)

	Features												
Transformation	F12	F13	F14	F15	F16	F17	F18	F19	F20	F21	F22	#P	|Avg|
adce	=	=	=	=	=	=↓	=↑	=↑	=	=	=	7	0.81
bdce	=	=	=	=	=	=↓	=↑	=↑	=	=	=	7	0.81
correlated-propagation	=	=	=	=	=	=	=	=↑	=	=	=	1	0.08
dse	=	=	=	=	=	=↓	=↑	=↑	=	=	=	7	0.04
early-cse	=	=	=	=	=↓-	↑↓	↑	=↑	=	=	=↓	17	7.91
globalopt	=	=	=	=	=	=↑	=↓	=↓	=↓	=	=	2	0.05
gvn	=	=	=	=	=↑↓	=↑↓	↓	↓	↓	=↑+	=↑↓	17	64.40
indvars	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	1.07
inline	=	=↑↓	=↑	=	=↑+	=↑	↑↓	↓↑	↑↓	=	=↑↓	17	8.54
instcombine	=	=	=	=	=↓	↑↓	↓↑	↓	=	=	=↓	17	88.71
ipsccp	=	=↓	=	=	=	=↓	=↓	=↑↓	=↓	=	=	11	0.06
jump-threading	=	↑	=↓	=	=↑	↑	↓	↓	↓	=↑+	↑	17	30.42
lcssa	=↑	=	=	=	=	=	=	=↓	=	=	=	4	0.58
licm	=	=↓↑	=	=	=	=↑↓	=↑	=↑	=↑	=	=↓	16	1.09
loop-deletion	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	1.07
loop-idiom	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	1.07
loop-rotate	=	=↑↓	=	=	=	↑	↑	↑	↑	=	=↓	17	10.88
loop-simplify	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	0.98
loop-unroll	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	1.02

Table 5: Continue

loop-unswitch	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	1.07l
oop-vectorize	=	=↑↓	=	=	=	=	=↑	=↑	=↑	=	=↓	16	1.07
mem2reg	=	=	=	=	=↓-	↓	↑	↓	↓	↑+	↓	17	222.72
memcpyopt	=	=	=	=	=	=↓	=↑	=	=	=	=	2	0.01
prune-eh	=	=↓	=	=	=	=↑	=↓	=↓	=↓	=	=	5	0.64
reassociate	=	=	=	=	=	=↑↓	=↑↓	=↑	=	=	=↓	14	0.63
sccp	=	=	=	=	=	=↓	=↑	=↑	=	=	=	3	0.01
simplifycfg	=	↑↓	=↓	=	=↑	↑↓	↓	↓	↓	=	↑	17	21.42
slp-vectorizer	=	=	=	=	=	=↑	=↓	=	=	=	=	1	0.03
sroa	=	=	=	=	=↓-	↓	↑	↓	↓	↑+	↓	17	225.13
tailcallelim	=	=↑↓	=	=	=	=↓	=↑	=↑	=↑	↑+	=	8	0.26

Table 6: Results for DNA-based features

Indels

Transformation	DNA	Matches	Mismatches	Insertions	Removals	#Programs
Adc	186916.14	186355.0	28.79	0.0	532.36	14
Bdce	186916.14	186354.0	28.79	0.0	533.36	14
correlated-propagation	1037375.0	1037371.0	0.0	0.0	4.0	1
dse	198823.85	198679.92	0.62	0.0	143.31	13
early-cse	155068.35	134456.71	854.71	10.59	19756.94	17
globalopt	199072.83	199063.42	0.08	12.58	9.33	12
gvn	155068.35	140944.0	4765.06	2804.76	9359.29	17
indvars	164382.81	164378.56	3.06	161.44	1.19	16
inline	155068.35	154760.53	307.06	61356.47	0.76	17
instcombine	155068.35	125791.06	13555.65	2367.88	15721.65	17
ipsccp	169402.47	169166.53	5.93	0.0	230.0	15
jump-threading	155068.35	149717.24	638.41	730.12	4712.71	17
lcssa	122665.00	122665.0	0.0	11.0	0.0	4
licm	164382.81	163978.44	68.19	467.13	336.19	16
loop-deletion	164382.81	164378.63	3.06	161.38	1.13	16
loop-idiom	164382.81	164378.63	3.06	161.38	1.13	16
loop-rotate	155068.35	153331.82	1512.82	3645.71	223.71	17
loop-simplify	164382.81	164378.63	3.06	158.63	1.13	16
loop-unroll	164382.81	164376.5	3.13	163.31	3.19	16
loop-unswitch	164382.81	164378.63	3.06	161.38	1.13	16
loop-vectorize	164382.81	164378.63	3.06	161.38	1.13	16
mem2reg	155068.35	98811.94	12866.29	537.29	43390.12	17
memcpyopt	227864.67	227805.67	37.22	0.0	21.78	9
prune-eh	115949.60	115321.4	401.4	345.2	226.8	5
reassociate	155068.35	154425.53	82.65	62.0	560.18	17
sccp	184486.23	184329.38	9.38	0.0	147.46	13
simplifycfg	155068.35	146342.76	2375.76	771.71	6349.82	17
slp-vectorizer	122814.00	122741.0	56.0	92.5	17.0	4
sroa	155068.35	98244.53	12977.41	559.47	43846.41	17
tailcallelim	248673.00	248663.13	0.38	18.88	9.5	8

Table 7: The patterns

Transformation	Exchange	Insertion	Removal
Adce	M⇒T M⇒M C⇒T O⇒T	-	T M C O
Bdce	M⇒T M⇒M C⇒T O⇒T	-	I M C O
correlated-propagation	-	-	O
dse	M⇒M O⇒T	-	T B M C O
early-cse	T⇒C T⇒B T⇒M T⇒O B⇒B B⇒O B⇒M B⇒I B)C B⇒T I⇒T I⇒B I⇒C I⇒I I⇒M I⇒O M⇒B M⇒C M⇒M M⇒O M⇒I M⇒T C⇒I C⇒O C⇒M C⇒B C⇒C C⇒T O⇒T O⇒B O⇒C O⇒O O⇒M	T B I M C O	T B I A M C O
Globalopt	M⇒M	M C O	M O
gvn	T⇒A T⇒C T⇒B T⇒I T⇒M T⇒O T⇒T B⇒C B⇒B B⇒O B⇒M B⇒I B⇒T I⇒C I⇒M I⇒I I⇒T I⇒O	T B I A M C O	T B I A M C O

Table 7: Continue

	A⇒T A⇒O M⇒A M⇒B M⇒C M⇒M M⇒O M⇒I M⇒T C⇒I C⇒O C⇒M C⇒B C⇒C C⇒A C⇒T O⇒T O⇒B O⇒C O⇒A O⇒O O⇒M O⇒I		
indvars	T⇒T	T C O	C O
inline	T⇒C T⇒B T⇒M T⇒O T⇒T B⇒O B⇒M B⇒C I⇒T I⇒C M⇒O M⇒T M⇒M M⇒C C⇒O C⇒M C⇒C O⇒T O⇒B O⇒C O⇒O O⇒M	T B I A M	C O T O
instcombine	T⇒C T⇒B T⇒I T⇒M T⇒O T⇒T B⇒C B⇒B B⇒O B⇒MB⇒I B⇒T I⇒B I⇒I I⇒MI⇒O I⇒T I⇒C A⇒T A⇒M M⇒B M⇒C M⇒M M⇒O M⇒I M⇒T C⇒I C⇒O C⇒MC⇒B C⇒C C⇒A C⇒T O⇒T O⇒B O⇒C O⇒O O⇒M O⇒I	T B I M C O	T B I A M C O
ipsccp	T⇒M T⇒O B⇒O M⇒T M⇒O O⇒T	-	T B I M C O
jump-threading	T⇒A T⇒B T⇒I T⇒M T⇒C T⇒O T⇒T B⇒M B⇒T I⇒C I⇒MI⇒O A⇒OM⇒BM⇒MM⇒OM⇒IM⇒T)C C)O C)M O)T O)C O)O O)M O)I	T B I A M C O	T B I A M C O
lcssa	-	O	-
licm	T⇒M T⇒C T⇒O T⇒T B⇒B B⇒M A⇒M M⇒B M⇒M M⇒O M⇒I M⇒T M⇒C C⇒A C⇒O C⇒M ⇒T O⇒T O⇒O O⇒M O⇒C	T I M C O	T B I M C O
loop-deletion	T⇒T	T O	O
loop-idiom	T⇒T	T O	O
loop-rotate	T⇒M T⇒O T⇒T I⇒T M⇒C M⇒M M⇒O M⇒I M⇒T C⇒T O⇒T O⇒O O⇒M O⇒I	T B I M C O	T M O
loop-simplify	T⇒T	T O	O
loop-unroll	T⇒T O⇒T	T M C O	T M C O
loop-unswitch	T⇒T	T O	O
loop-vectorize	T⇒T	T O	O
mem2reg	T⇒C T⇒B T⇒I T⇒M T⇒O T⇒T B⇒C B⇒B B⇒A B ⇒O B⇒MB⇒I B⇒T I⇒B I⇒I I)MI⇒O I)T I⇒C A⇒T A⇒O M⇒A M⇒B M⇒C M⇒M M⇒O M⇒I M⇒T C⇒I C⇒O C⇒M C⇒B C⇒C C⇒T O⇒T O⇒B O⇒C O⇒O O⇒M O⇒I	T B I M C O	T B I A M C O
memcpyopt	M⇒M M)O M⇒C C⇒O O⇒M		M C O
prune-eh	T⇒M T⇒C T⇒O T⇒T A⇒T A⇒O M⇒O M⇒T C⇒O C⇒T O⇒T O⇒O O⇒M	T M C O	T A M C O
reassociate	T⇒B T⇒O B⇒C B⇒B B⇒O B⇒M I⇒B I⇒M M⇒B ⇒C M⇒M M⇒O M⇒I M⇒T C⇒B C⇒T O⇒T O⇒B	B I M	T B I M C O
sccp	B⇒T I⇒T M⇒T M⇒O C⇒T O⇒		T T B I M C O
simplifycfg	T⇒C T⇒B T⇒I T⇒M T⇒O T⇒T B⇒B B⇒O B⇒M ⇒T B⇒C I⇒T I⇒M I⇒I I⇒O I⇒C A⇒T A⇒M A⇒O A⇒A A⇒C M⇒A M⇒B M⇒C M⇒M M⇒O M⇒I M⇒T C⇒I C⇒O C⇒M C⇒C C⇒T O⇒T O⇒B O⇒C O⇒A O⇒O O⇒M O⇒I	T B I A M C O	T B I A M C O
slp-vectorizer	B⇒M B⇒V M⇒B M⇒C M⇒M M⇒V C⇒M O⇒C O⇒M	B V M C	B M C
Sroa	T⇒A T⇒C T⇒B T⇒I T⇒M T⇒O T⇒T B⇒C B⇒B B⇒A B⇒O B⇒M B⇒I B⇒V B⇒T I⇒B I⇒I I)M I⇒O I⇒T I⇒C A⇒T A⇒M A⇒O A⇒C M⇒B M⇒C M⇒M M⇒O M⇒I M⇒T M⇒V M⇒A C⇒I C⇒O C⇒MC⇒B C⇒C C⇒A C⇒V C⇒T O⇒V O⇒T O⇒B O⇒C O⇒A O⇒M O⇒O O⇒I	T B I A M C O	T B I A M C O
tailcallelim	T⇒T	T O	O

T = terminator, B = binary, I = bitwise, V = vector, A = aggregate, M = memory, C = conversion, O = other

DOI: 10.3844/jcssp.2019.1678.1693

Table 8: The improvements

	General			Only Improvement			
Transformation	Min	Avg	Max	Min	Avg	Max	#Programs
O1	-	-	-	18.76	73.52	165.27	17
O2	-	-	-	16.89	95.02	194.38	17
O3	-	-	-	17.09	94.20	192.13	17
adce	-10.48	-3.02	2.19	0.18	1.06	2.19	3
bdce	-9.20	-3.30	1.31	1.03	1.20	1.31	3
correlated-propagation	-10.17	-3.33	0.94	0.73	0.84	0.94	3
dse	-10.01	-3.62	3.03	0.59	1.81	3.03	2
early-cse	-3.54	0.64	9.67	0.01	2.50	9.67	11
globalopt	-12.60	-4.02	4.03	0.27	1.74	4.03	3
gvn	-3.31	4.23	23.40	0.01	7.13	23.40	11
indvars	-11.05	-3.92	4.54	1.17	2.35	4.54	3
inline	-11.23	0.09	18.35	0.61	10.43	18.35	6
instcombine	-9.76	3.00	14.00	0.28	6.11	14.00	12
ipsccp	-9.61	-4.07	4.25	0.44	1.71	4.25	3
jump-threading	-10.73	-3.54	5.44	0.04	2.60	5.44	4
lcssa	-13.42	-4.92	4.88	4.88	4.88	4.88	1
licm	-10.77	-4.54	5.11	1.01	2.87	5.11	3
loop-deletion	-10.02	-4.46	4.55	1.03	2.30	4.55	3
loop-idiom	-11.34	-4.19	4.55	0.14	1.83	4.55	3
loop-rotate	-8.86	-2.64	5.58	0.26	2.21	5.58	6
loop-simplify	-10.29	-4.65	3.72	0.75	2.24	3.72	2
loop-unroll	-10.68	-4.43	3.75	1.12	2.44	3.75	2
loop-unswitch	-12.34	-4.81	2.97	0.80	1.89	2.97	2
loop-vectorize	-14.54	-5.44	3.24	3.24	3.24	3.24	1
mem2reg	16.06	55.25	116.14	16.06	55.25	116.14	17
memcpyopt	-14.48	-4.92	3.78	0.38	2.08	3.78	2
prune-eh	-11.83	-4.43	1.13	0.59	0.92	1.13	3
reassociate	-13.46	-5.17	1.72	0.19	0.95	1.72	2
sccp	-10.57	-3.54	1.51	0.13	1.01	1.51	3
simplifycfg	-11.06	-3.94	1.7	0.09	0.83	1.70	4
slp-vectorizer	-10.66	-3.15	2.42	0.29	1.41	2.42	4
sroa	16.42	57.17	125.44	16.42	57.17	125.44	17
tailcallelim	-11.17	-3.28	3.34	0.14	1.89	3.34	5

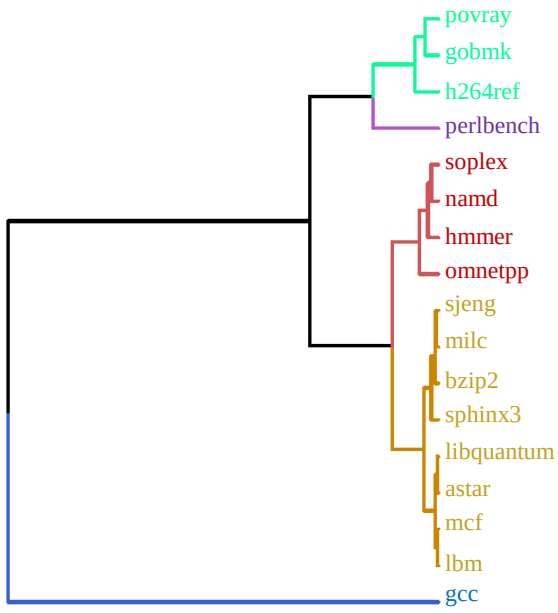


Fig. 1: Clustering

DOI: 10.3844/jcssp.2019.1678.1693

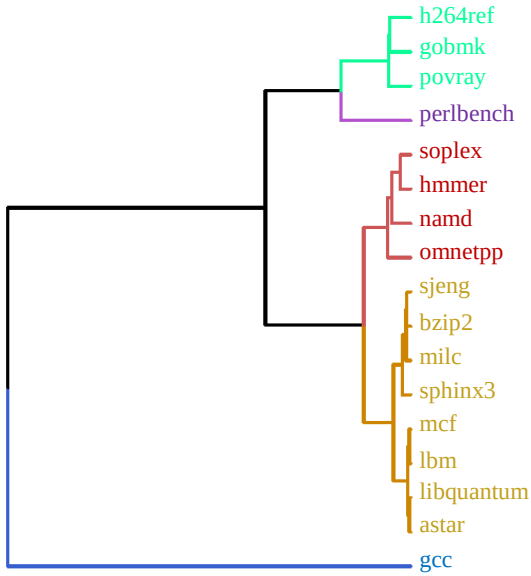


Fig. 2: Clustering

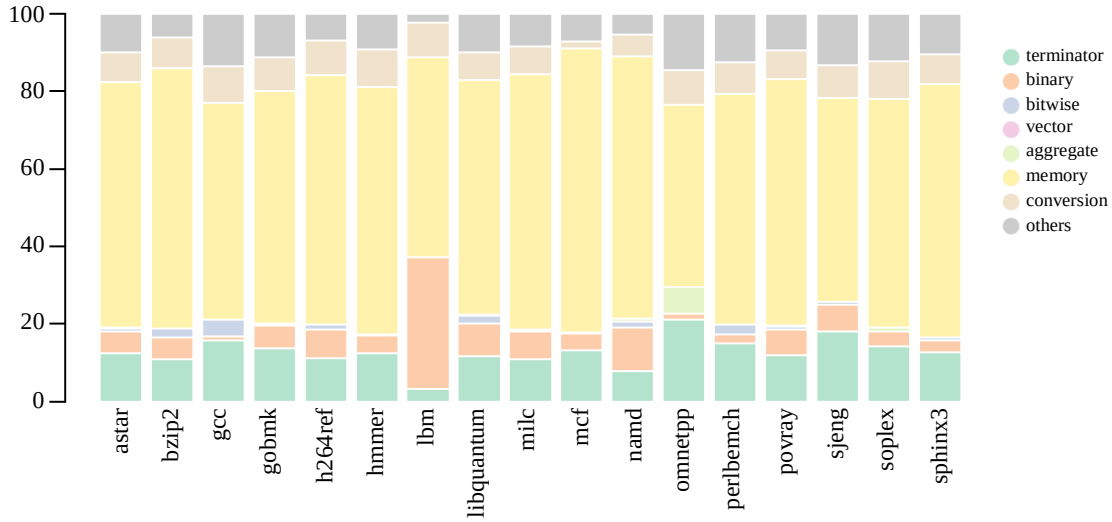


Fig. 3: The distribuition of instructions

Evaluating the DNA-Based Features

Diversity

Again, we will go into the diversity between programs, however, according to the instructions that compose it. Figure 2 displays the clusters of programs.

Clustering the programs based on DNA-based features (their instructions) follows the same distribution of clustering based on CFG-based features. PERLBENCH and GCC are the two programs highlight, the clusters remain unbalanced and only two clusters comprise 70% of the programs. However, there are two changes related to smaller clusters. First, H264REF forms a cluster with GOBMK and not with POVRAY. Second, SOPLEX forms a cluster with HMMER and not with NAMD.

An Overview of the Programs

A representation based on IR instructions makes it possible to analyze the distribution of the instructions that compose the programs. Figure 3 shows the distribution of instructions according to their classes, without using a TA.

The composition of the programs is dominated by memory instructions. These instructions consume from 46.91% to 73.38% of total instructions. The second class that dominates most programs (15 in the case) is the terminator instructions, with a percentage ranging from 10.81% to 21.03%. Only the programs LBM and NAMD

have the second dominant class the binary operations, whose percentages are 33.80% and 11.24%, respectively.

Excluding LBM and NAMD, binary instructions do not compose more than 10% of all instructions. This pattern is followed by the conversion and other instructions. The other classes do not consume more than 4% of the instructions.

TAs that reduce the number of memory instructions have great potential in improving the performance of programs.

Our Findings

Table 6 presents the results obtained using the DNA-based features. This table displays the length of the DNA (without using a TA during target code generation), the number of matches, mismatches and indels, besides the number of programs affected by the TA. As in the previous section, only the results for those TAsthat affect programs are displayed.

Twenty three (23) TAs, out of 30, have the ability to exchange, insert or remove instructions. The other seven (adce, bdce, correlated-propagation, dse, ipsccp, memcpyopt and sccp) do not have the ability to insert instructions.

Related to the change rate in a DNA, some TAs stand out among others. The TAs gvn, instcombine, mem2reg and sroa have a superior potential in exchanging instructions when compared with other TAs. Regarding the insertion of instructions, no TA compares to inline. This was already expected, by the nature of this transformation that exchanges a function call for its body.

In terms of IR change rate, it is difficult to determine which would be an ideal rate without analyzing the classes of the instructions and consequently the correspondent machine instructions (assembly). Remember each assembly instruction has a cost, which is determined in clock cycles. In this way, to improve program performance (reduce its runtime) the ideal is to reduce the number of assembly instructions in the target code and preferably reduce those instructions that have a high cost (memory access instructions). This indicates that although Table 6 presents theoretically low rates, these can have a significant impact on the performance of the target code. This will be shown in the next section.

Regardless of the change rate, our goal is understand how TAs affect IR. In this context, we can notice that different TAs operate in different degrees, but usually always performing three distinct actions (remove, insert and exchange). It is also clear, from the results, although TAs have the potential to do three distinct actions, one action always dominates the others. This means, TAsare classified as follows:

1. TAs whose predominant action is to insert instructions: global-opt, ind-vars, inline, lcssa, licm loop-deletion, loop-idiom, loop-rotate, loop-simplify loop-unroll, loop-unswitch loop-vectorize, slp-vectorizer and tailcallelim
2. TAs whose predominant action is to remove instructions: adce, bdce correlated-propagation, dse early-cse, gvn, instcombine, ipsccp, jump-threading, mem2reg, reassociate, sccp, simplifycfg and sroa
3. TAs whose predominant action is to exchange instructions: memcpyopt and prune-eh

Several TAs affect most programs, only four TAs impact less than 50% of the programs (correlated-propagation, lcssa, prune-eh and slp-vectorizer).

Comparing the results obtained by the two different features, it is possible to notice a difference in the number of programs affected by some TA. Analyzing the instructions that compose a program, it is possible to realize a greater amount of TAs affects the programs and not the amount specified by an analysis based only on the CFG of the program.

The analysis based on DNA-based features indicates adce, bdce affect 82% and not only 4% of the evaluated programs; globalopt affects 71% and not 12%; ipsccp affects 88% and not 65%; memcpyopt affects 53% and not 12%; reassociate affects 100% and not 82%; sccp affects 76% and not 18%; and slp-vectorizer affects 24% and not 5%. This is due to the CFG-based features is a macro representation, compared to the DNA-based features.

Table 7 shows the patterns in the transformation of a DNA. In this table, the second column presents the source and target class in an instruction exchange. The third and fourth columns display the classes of the instructions, which are inserted and removed.

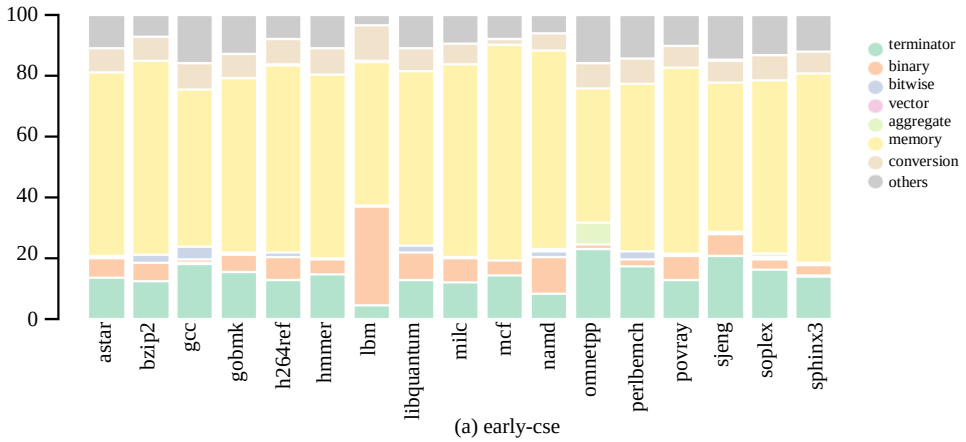
Several TAs operate on several classes of instructions, such as early-cse, gvn, inline, instcombine, jump-threading, licm, loop-rotate, mem2reg, reassociate, simplifycfg and sroa. Only sroa operates with all classes of instructions. In addition, a few TAs follow the same pattern, namely, loop-deletion loop-idiom, loop-simplify loop-unswitch, loop-vectorize and tailcallelim. These are the TAs with the smallest scope of action.

Two TAs handle vector instructions, slp-vectorize, as expected, as well as sroa. However, while sroa only exchange some instructions for vector instructions, slp-vectorize changes vector instructions as well as inserts new vector instructions in the IR.

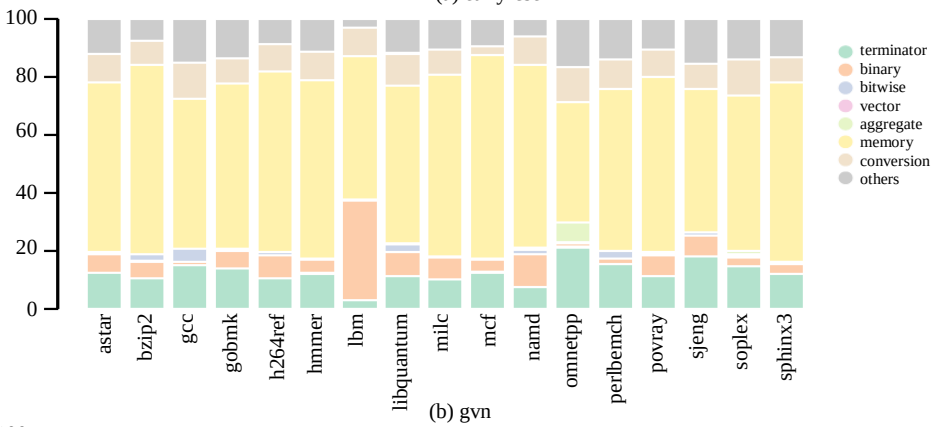
Improvement

Finally, the last question analyzed is the performance of the TAs related to reducing the program runtime.

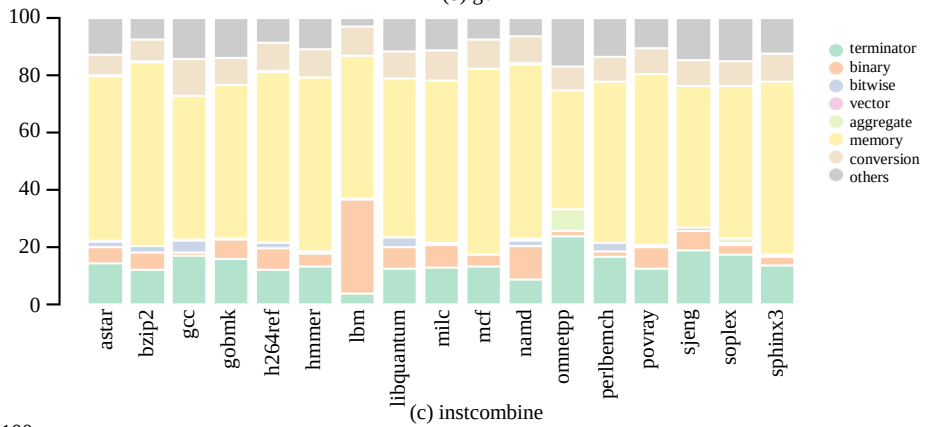
DOI: 10.3844/jcssp.2019.1678.1693



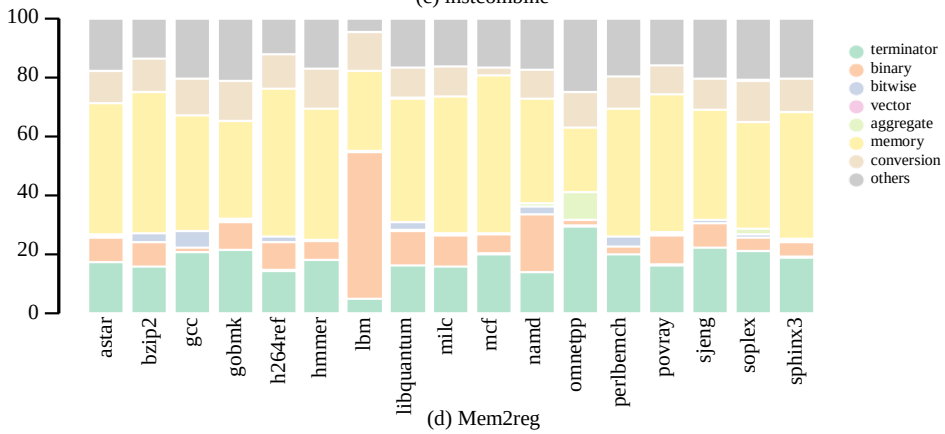
(a) early-cse



(b) gvn



(c) instcombine



(d) Mem2reg

DOI: 10.3844/jcssp.2019.1678.1693

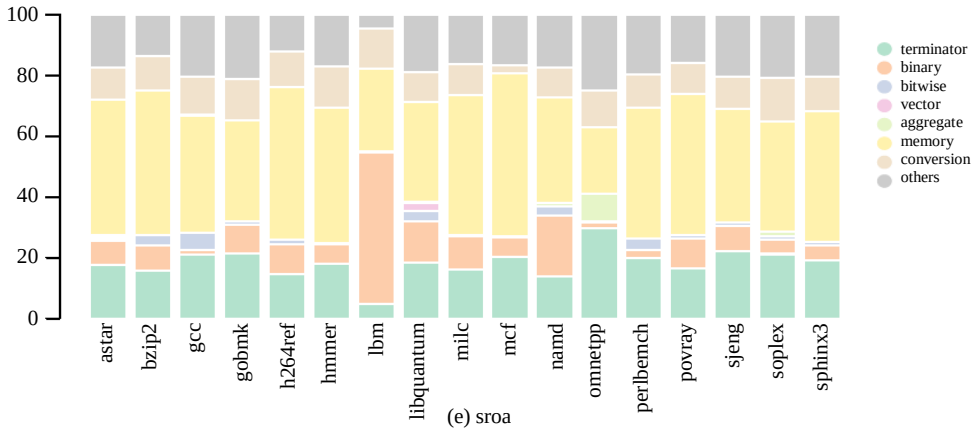


Fig. 4: The distribution of instruction

This analysis is essential to corroborate the results presented in the previous sections.

Table 8 shows the improvement performed by each TA and the three compiler optimization levels. This table presents minimum, maximum and average improvement, besides presents the same metrics considering only programs which have performance gain.

Several TAs (25) cause an average loss of performance. On the other hand, considering only the TAs which have achieved a performance gain, the scenario is different. In this case, the average performance gain is up to 3%, but for no more than 4 programs. Inline is an exception to this case. Although, inline does not achieve an average performance considering the total programs, inline achieves a performance gain of 10.43%, when it is considered the programs that achieve performance gain.

It is evident that although several TAs change the characteristics of the IR, such changes may have an undesired effect. As mentioned before, although such algorithms are considered optimizations there is no guarantee of performance gain. Due to this fact, some researchers prefer to use the term transformation rather than optimization.

The most effective TAs are early-cse, gvn, instcombine, mem2reg and sroa, both in terms of performance gain achieved and field of activity. The average performance gain achieved by the first three TAs reaches 4.23% overall. Whereas for the subset of programs with performance gain, the average gain reaches 7.30% for a universe of 12 out of 17 programs.

The two excellent TAs are mem2reg and sroa. These two can achieve a better performance gain than all other TAsand are capable of improving the performance of all programs in the universe of the programs evaluated. For mem2reg and sroa the performance gain reaches 57.17% in the overall average.

The compiler optimization levels achieve performance gain for all programs, because they are formed by a sequence of TAs. For the universe of TAs evaluated, the performance gain is very far from that achieved by a compiler optimization level, except for mem2reg and sroa.

The distances between the average performance obtained by mem2reg and the compiler optimizations levels are 24.81%, 41.85%, 41.35%, respectively for O1, O2and O3 and the distances for sroa are 22.24%, 39.83%, 39.31%, respectively for O1, O2and O3.

An important issue is to clarify what led early-cse, gvn, instcombine, mem2reg and sroa to perform better than the other TAs. In order to clarify this fact, Fig. 4 presents the decomposition of the instructions that compose the programs, for these five TAs.

The performance gain obtained by these TAs was due to the reduction of memory instructions. Such reduction follows a ratio of 4.10%, 5.30%, 6.15%, 24.22% and 24.32%, respectively for early-cse, gvn, instcombine, mem2reg and sroa. The other classes of instructions had an increase, which is more accentuated for mem2reg and sroa.

Conclusion

Modern compilers provide several code transformation algorithms to be applied to the source code, during target code generation, which goal is to improve the quality of the target code. Such algorithms transform the code structure, usually at the intermediate representation level, without altering the semantics of the program.

Although the goal of a transformation is to improve the quality of the target code, in practice this does not always occur. In fact, a particular transformation can degrade the performance of a program. This phenomenon is due to the changes made in the program negatively affect the components of the hardware in question, such as the misuse of memory.

In this article, we show how several code transformation algorithms modify the structure of the program, at the intermediate representation level. In addition, we show code transformation algorithms have a

direct impact on the reduction of program runtime, when instructions are removed from the target code, mainly memory instructions.

As future work, we plan to create a model to infer what code transformation algorithm the compiler should enable during target code generation, based on our findings in this article.

Acknowledgement

This work manuscript undergoing several modifications before the final version was submitted. The corresponding author would like to thanks the co-author and the reviewers.

Author's Contributions

Anderson Faustino da Silva: Proposes the idea of this researches, besides performing the experiments and writing the text

Leonardo Daganello de Souza: Creates a system to evaluate the data, besides performing some analysis and creating figures

Ethics

This article is an original research and contains unpublished material. The authors confirm that there is no conflict of interest involved.

References

Aho, A.V., M.S. Lam, R. Sethi and J.D. Ullman, 2006. Compilers: Principles, techniques and tools. Prentice Hall.

Corporation, S.P.E., 2019.

De Lima, E.D., A.F. da Silva and C. Herrera, 2013a. A case-based reasoning approach to find good compiler optimization sequences. Proceedings of the International Conference Chilean Computer Science Society, Nov. 11-15, IEEE Xplore Press, Temuco, Chile, pp: 8-10. DOI: 10.1109/SCCC.2013.21

De Lima, E.D., T.C. de Souza Xavier, A.F. da Silva and L.B. Ruiz, 2013b. Compiling for performance and power efficiency. Proceedings of the International Workshop Power Timing Modeling, Optimization Simulation, Sept. 9-11, IEEE Xplore Press, Karlsruhe, Germany, pp: 142-149. DOI: 10.1109/PATMOS.2013.6662167

Dong, S., O. Olivo, L. Zhang and S. Khurshid, 2015. Studying the influence of standard compiler optimizations on symbolic execution. Proceedings of the International Symposium Software Reliability Engineering, Nov. 2-5, IEEE Xplore Press, Gaithersbury, MD, USA, pp: 205-215. DOI: 10.1109/ISSRE.2015.7381814

Eyerman, S., L. Eeckhout and J.E. Smith, 2008. Studying compiler optimizations on superscalar processors through interval analysis. Proceedings of the 3rd International Conference High Performance Embedded Architectures Compilers, (EAC'08), Berlin, Heidelberg, pp: 114-129.

Filho, J.F., L.G.A. Rodriguez and A.F. da Silva, 2018. Yet another intelligent code-generating system: A flexible and low-cost solution. J. Comput. Sci. Technol., 33: 940-965. DOI: 10.1007/s11390-018-1867-7

Foleiss, J.H., A.F.D. Silva and L.B. Ruiz, 2011. The effect of combining compiler optimizations on code size. Proceedings of the 30th International Conference Chilean Computer Science Society, Nov. 9-11, IEEE Xplore Press, Curico, Chile, pp: 187-194. DOI: 10.1109/SCCC.2011.25

Hariri, F., A. Shi, H. Converse, S. Khurshid and D. Marinov, 2016. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. Proceedings of the 27th International Symposium Software Reliability Engineering, Oct. 23-27, IEEE Xplore Press, Ottawa, ON, Canada, pp: 105-115. DOI: 10.1109/ISSRE.2016.51

Henning, J.L., 2006. Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News, 34: 1-17. DOI: 10.1145/1186736.1186737

Ibrahim, M.E.A., M. Rupp and S.E. Habib, 2009. Compiler-based optimizations impact on embedded software power consumption. Proceedings of the Joint IEEE North- East Workshop Circuits Systems TAISA Conference, Jun. 28-Jul. 1, IEEE Xplore Press, Toulouse, France pp: 1-4. DOI: 10.1109/NEWCAS.2009.5290480

Lattner, C. and V. Adve, 2004. LLVM: A compilation framework for lifelong program analysis and transformation. Proceedings of the International Symposium Code Generation Optimization, Mar. 20-24, IEEE Xplore Press, USA, pp: 75-86. DOI: 10.1109/CGO.2004.1281665

Lee, C., J. Chang and R. Chang, 2012. Compiler optimization to reduce cache power with victim cache. Proceedings of the 9th International Conference Ubiquitous Intelligence Computing Autonomic Trusted Computing, Sept. 4-7, IEEE Xplore Press, Fukuoka, Japan, pp: 841-844. DOI: 10.1109/UIC-ATC.2012.36

Muchnick, S.S., 1997. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Namolaru, M., A. Cohen, G. Fursin, A. Zaks and A. Freund, 2010. Practical aggregation of semantical program properties for machine learning based optimization. Proceedings of the International Conference Compilers, Architectures Synthesis Embedded Systems, Oct. 24-29, Scottsdale, AZ, USA. ACM, pp: 197-206. DOI: 10.1145/1878921.1878951

Needleman, S.B. and C.D. Wunsch, 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Molecular Biol., 48: 443-453. DOI: 10.1016/0022- 2836(70)90057-4

Queiroz Junior, N.L. and A.F. da Silva, 2015. Finding good compiler optimization sets - a case-based reasoning approach. Proceedings of the 17th International Conference Enterprise Information Systems, (EIS'15), Barcelona, Spain, pp: 504-515. DOI: 10.5220/0005380605040515

Scott, M.L., 2009. Programming language pragmatics. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Sebesta, R.W., 2009. Concepts of programming languages. Addison Wesley, San Francisco, CA, USA.

Seng, J.S. and D.M. Tullsen, 2003. The effect of compiler optimizations on Pentium 4 power consumption. Proceedings of the Seventh Workshop Interaction Between Compilers Computer Architectures, Feb. 8-8, IEEE Xplore Press, Anaheim, CA, USA, pp: 51-56. DOI: 10.1109/INTERA.2003.1192355

Siraichi, M.Y., C. Tonetti and A.F. da Silva, 2016. A design space exploration of compiler optimizations guided by hot functions. Proceedings of the International Conference Chilean Computer Science Society, Oct. 10-14, IEEE Xplore press, Valparaiso, Chile, pp: 1-12. DOI: 10.1109/SCCC.2016.7836038

Tartara, M. and S. Crespi Reghizzi, 2013. Continuous learning of compiler heuristics. ACM Trans. Architecture Code Optimzation, 9: 1-46. DOI: 10.1145/2400682.2400705

Team, L., 2019. The llvm compiler infrastructure.

Yuan, S., P. Lin, C. Su and T.H Chen, 2018. Compiler options effect on system-level near field EMI. Proceedings of the International Symposium Electromagnetic Compatibility Asia-Pacific Symposium Electromagnetic Compatibility, May 14-18, IEEE Xplore Press, Singapore, Singapore, pp: 848-851. DOI: 10.1109/ISEMC.2018.8393901