Journal of Universal Computer Science, vol. 25, no. 1 (2019), 42-72
submitted: 20/1/17, accepted: 28/12/18, appeared: 28/1/19 © J.UCS

Pinhão: An Auto-tunning System for Compiler Optimizations Guided by Hot Functions

Marcos Yukio Siraichi, Caio Henrique Segawa Tonetti
Anderson Faustino da Silva
(State University of Maringá, Maringá, Brazil
sir.yukio@gmail.com, caio.tonetti@gmail.com, anderson@din.uem.br)

Abstract: The literature presents several auto-tunning systems for compiler optimizations, which employ a variety of techniques; however, most systems do not explore the premise that a large amount of program runtime is spent by hot functions which are the portions at which compiler optimizations will provide the greatest benefit. In this paper, we propose `Pinhão`, an auto-tunning system for compiler optimizations that uses hot functions to guide the process of exploring which compiler optimizations should be enabled during target code generation. `Pinhão` employs a hybrid technique – a machine learning technique, as well as an iterative compilation technique – to find an effective compiler optimization sequence that fits the characteristics of the unseen program. We implemented `Pinhão` as a `LLVM` tool, and the experimental results indicate that `Pinhão` finds effective sequences evaluating a few points in the search space. Furthermore, `Pinhão` outperforms the well-engineered compiler optimization levels, as well as other techniques.

Key Words: Auto-tunning system, compiler, optimization, machine learning, iterative compilation
Category: D.3.4

1 Introduction

Modern compilers [Cooper and Torczon, 2011] provide several optimizations (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 optimizations should be turned on or off. To address this issue, modern compilers provide several compiler optimization sequences[1], known as compiler optimization levels.

The first-generation auto-tunning systems, whose goal is to find an effective sequence[2], employ the technique known as iterative compilation [Park et al., 2011, Purini and Jain, 2013]. This means that such systems evaluate[3] several sequences, and return the best target code. Due to the diversity of possible sequences, these systems tries to cover the search space selectively.

[1] Compiler optimization sequence will be cited as sequence.
[2] An effective sequence is a sequence that when enabled during target code generation provides performance concerning the the desired goal – for example, to reduce the runtime – surpassing a threshold.
[3] Evaluating a sequence means compiling a program using this sequence and measuring its runtime.

The second-generation auto-tunning systems employ machine learning techniques [Agakov et al., 2006, Cavazos et al., 2007, Kulkarni and Cavazos, 2012], in order to minimize the number of sequences evaluated. In such systems, the goal is to learn from past experiences and so using these experiences to find an effective sequence.

Although, it is known that a large amount of program runtime is spent in hot functions, neither iterative compilation nor machine learning techniques, presented in the literature, take into account such functions to guide the process of exploring sequences. In fact, hot functions are the portions at which compiler optimizations will provide the greatest benefit.

In this paper, we propose an auto-tunning system for compiler optimizations, called `Pinhão`[4], which takes into account hot functions.

The motivation in developing `Pinhão` is based on three premises. First, it is possible to find similar patterns among programs, which give important insights for exploring potential sequences. Second, similar programs react approximately equal, when they are compiled using the same sequence. Third, hot functions can provide important insights at which sequence the code generator should enable.

`Pinhão` is classified as a hybrid technique, because it employs a machine learning technique, as well as an iterative compilation technique. Employing a machine learning technique indicates that `Pinhão` tries to solve a new problem using a solution of an previous similar problem. Basically, training and test programs are represented by feature vectors in a multidimensional space and a similarity model operates in this space trying to find similar points, which indicates similar patterns that should be exploited.

It is possible that previous solutions do not fit the requirements of new problems. In other words, the characteristics of a previously-compiled program do not fit the characteristics of the new program. Thus, by employing an iterative compilation technique `Pinhão` tries to find an effective sequence, if the machine learning technique fails.

We implemented `Pinhão` as a tool of `LLVM` infrastructure [LLVM Team, 2016], and evaluated `Pinhão` on different configurations. The experimental results indicate that `Pinhão` surpasses the performance obtained by the well-engineered compiler optimization levels, as well as other techniques.

The rest of this paper is organized as follows. Section 2 details our auto-tunning system, `Pinhão`. As `Pinhão` relies on a database of previously-generated sequences, Section 3 describes how we created this database. Section 4 describes the experimental setup and the methodology used in the experiments, which is followed by the experimental results in Section 5. In Section 6 we survey some related work. Finally, Section 7 summarizes this paper and provides some

[4] Pinhão is the seed of *Araucaria angustifolia*, a tree of great importance in the southern and southeastern regions of Brazil.

insights for future work.

2 Pinhão: An Auto-tunning System for Compiler Optimizations

During compilation, the compiler applies several optimizations to improve the target code quality; however, some optimizations can be useful to a specific program, but not to another. Thus, the most appropriate approach is to choose optimizations considering that it is a program-dependent problem. To address this problem, this section presents Pinhão, an auto-tunning system for compiler optimizations guided by hot functions.

Figure 1 depicts `Pinhão`'s workflow.

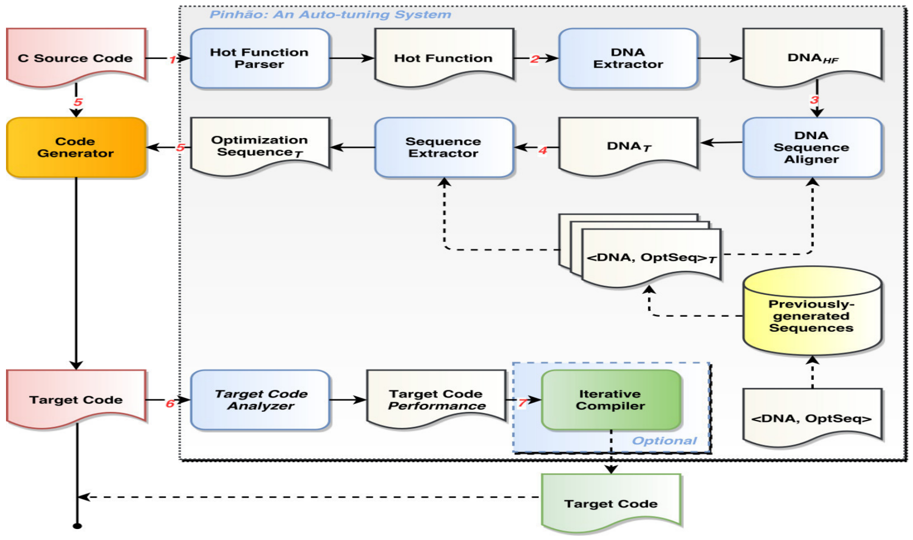


Figure 1: Workflow for Finding a Sequence

First, `Pinhão` extracts the program's hottest function, which will be represented by a symbolic representation, similar to a `DNA`. This `DNA` will be aligned with the previously-generated `DNAs` storage in a database, in order to find similar previously-compiled programs. In the following step, `Pinhão` extracts from the database N sequences. At this point, `Pinhão` invokes the code generating system to evaluate such sequences. Which means that `Pinhão` will compile the whole test program using each one of the N sequences, and measure the performance (runtime) of each target code. After generating the target code, `Pinhão` either returns the best target code or invokes an iterative compiler. Invoking an

`Iterative Compiler` (`IC`) is an optional step, and it is performed if and only if the performance of the target code is not better than a threshold (the performance of `LLVM` compiler optimization levels). Finally, the last step is to update the database with new knowledge.

`Pinhão` tries to find similar patterns among programs focusing on which compiler optimizations will provide the greatest benefit when applied to hot functions. In fact, `Pinhão` can focus on only one or several hot functions. It is important to note that a program generally consists of several source codes (modules); therefore each one has its hot function. If `Pinhão` is not able to find an effective sequence after evaluating hot functions, it may call the `Iterative Compiler`. We describe below the four possible different behaviours:

1. `Pinhão.H`: this configuration inspects only one hot function: the program's hottest function. The whole program will be compiled using the sequence that fits the characteristics of its hottest function. It does not invoke the `IC`.

2. `Pinhão.MH`: this configuration inspects several hot functions. Each module is compiled using the sequence that fits the characteristics of its hot function. In addition, it turns off the `IC`.

3. `Pinhão.H+IC`: this configuration is similar to `Pinhão.H`, except that it turns on the `IC`.

4. `Pinhão.MH+IC`: this configuration is similar to `Pinhão.MH`, except that it turns on the `IC`.

The two first configurations (`Pinhão.H` and `Pinhão.MH`) are pure machine learning techniques, and the last two (`Pinhão.H+IC` and `Pinhão.MH+IC`) are hybrid techniques. `Pinhão.H` and `Pinhão.MH` are appealing techniques because it reduces the response time using a model that provides insights on what sequence the compiler should enable during target code generation. This means that such techniques inspect few points in the search spaces. On the other hand, the hybrid techniques are time-consuming as they inspect several points in the search spaces.

Aiming for diversity, we propose the use of multiple hot functions (`MH`). As programs are composed of several modules (source code), each one will be compiled using its hot function. Then, the code generator links the object codes and finally generates the target code.

2.1 Hot Function Parser

The `Hot Function Parser` is the static analysis profiler proposed in [Wu and Larus, 1994], which estimates the relative execution frequency of pieces of the program, such as calls, procedure invocations, basic blocks, and control-flow edges. Such profiler is an appealing tool, due to not requiring neither program instrumentation nor execution.

Given the control-flow graph G for the function F, `Wu-Larus Static Analysis Profiler` performs two steps:

1. Calculate for each edge its probability; and
2. Transverse the control-flow graph propagating the probabilities.

In the first step, the profiler starts by combining the branch prediction heuristics proposed in [Ball and Larus, 1993], which are simple assumptions that take into account specific compiler implementations and architecture designs. Namely, there are seven of them:

1. **OpCode Heuristic:** if the branch condition is either a comparison of "less than zero" or "less than or equal zero," this branch will not be taken.
2. **Loop Heuristic:** if the successor is either a loop head or a loop pre-header, this branch will be taken.
3. **Pointer Heuristic:** if the branch condition is a comparison between two pointers, this branch will not be taken.
4. **Call Heuristic:** if the successor either contains a call or unconditionally reaches a block with a call, the other branch will be taken.
5. **Return Heuristic:** if the successor either contains a return or unconditionally reaches a block with a return, the other branch will be taken.
6. **Guard Heuristic:** if a register is an operand of this branch and it is used in the successor before it is defined, this branch will be taken.
7. **Store Heuristic:** if the successor has a store operation, the other branch will be taken.

These heuristics are only applied to what they call "non-loop branch", which consist in branches whose outgoing edges are neither exit edges nor backedges. As these heuristics are binary predictions, `Wu-Larus Static Analysis Profiler` relies on the experiments excuted by Ball and Larus and employed the frequency, at which the predictions were correct, as the branch probabilities. Table 1 presents these probabilities.

With the branch probabilities calculated, the second step takes place. In this step, these probabilities are propagated throughout the basic blocks, yielding the edge and basic block's frequencies.

Table 1: Branch probability of each heuristic [Ball and Larus, 1993]

Heuristic	Branch Probability
OpCode Heuristic	84%
Loop Heuristic	88%
Pointer Heuristic	60%
Call Heuristic	78%
Return Heuristic	72%
Guard Heuristic	62%
Store Heuristic	55%

Let $\text{freq}(b_i)$ be the frequency of the basic block i and let $\text{freq}(b_i \to b_j)$ be the frequency of the edge from basic block i to basic block j. Edge and basic block's frequencies are calculated as follows.

$$\text{freq}(b_i) = \begin{cases} 1 & \textit{if } b_1 \textit{ is the entry block} \\ \sum_{b_p \in \text{pred}(b_i)} \text{freq}(b_p \to b_i) & \textit{otherwise} \end{cases} \quad (1)$$

$$\text{freq}(b_p \to b_i) = \text{freq}(b_p) \times \text{prob}(b_p \to b_i) \quad (2)$$

As the Equation 1 shows, the frequency of the basic block i is calculated by the sum of all the edge frequencies from its predecessor, with exception of the entry basic block which frequency is 1 (one). The Equation 2 uses the probability calculated in step one and calculates the edge probability.

For functions that have loops, these equations become mutually recursive, turning the algorithm too slow and unable to handle loops with no apparent boundaries. To address this issue, Wu and Larus present an elimination algorithm as follows.

$$\text{cp}(b_0) = \sum_{i=1}^{k} r_i \times \text{prob}(b_i \to b_0) \quad (3)$$

$$\text{freq}(b_0) = \text{in\_freq}(b_0) + \sum_{i=1}^{k} \text{freq}(b_i \to b_0) = \frac{\text{in\_freq}(b_0)}{1 - \text{cp}(b_0)} \quad (4)$$

where b_0 is the loop header, $\text{cp}(b_0)$ is the cyclic probability and $\text{in\_freq}(b_0)$ is the incoming edge frequency. In the Equation 3, r_i represents the probability of the control flow from b_0 to b_i. Therefore, its multiplication with the probability of the branch represents the probability of taking the backedge from basic block b_i. The Equation 4 makes use of the cyclic probability to calculate the total basic block frequency of the loop header.

By going through these two steps, it is possible to calculate an estimation of a function's total cost. This process is illustrated by Equation 5, where for each function *f* its cost is calculated by summing the product of the basic block frequency ($\text{freq}(bb)$) by the cost of each instruction ($\text{cost}(i)$), for each basic block inside the function. Hence, these steps are applied to all functions and identify hot functions.

$$\text{cost}(f) = \sum_{bb \in f} \sum_{i \in bb} \text{cost}(i) \times \text{freq}(bb) \quad (5)$$

It is important to note that the approach proposed by Wu and Larus always assigns each function a score, which indicates its weight. As a result, `Pinhão` can always extract the hot function. In case of ties, `Pinhão` uses a random strategy to select one function.

2.2 Representing Hot Functions

Machine learning techniques rely on exposing the similarities among programs to identify patterns and decide what sequence should be enabled during target code generation.

Previous researchers represented programs using:

- performance counters [Cavazos et al., 2007];
- control-flow graphs [Park et al., 2012];
- compilation data [Queiroz Junior and da Silva, 2015];
- numerical features [Namolaru et al., 2010, Tartara and Reghizzi, 2013]; or
- a symbolic representation [Sanches and Cardoso, 2010, Martins et al., 2014].

Performance counters are dynamic characteristics that describe the program behavior in regards to its execution. The others are static characteristics that describe the algorithmic structures of the program. The appeal of dynamic characteristics is that it considers both the program and hardware characteristics. However, dynamic characteristics provide a disadvantage due to being platform-dependent and, thus, incurring the need for program execution. Alternatively, static characteristics are platform-independent and do not require program execution. However, such representation does not consider the program-input data, which is an element that can alter the program's behavior and consequently cause parameter alterations of the code-generating system.

In this work, we use static characteristics to represent programs. Such representation is a symbolic representation, similar to a `DNA`, which encodes program elements into a single string. Our proposal differs from previous work [Sanches and Cardoso, 2010, Martins et al., 2014], due to we apply transformation rules on intermediate code, instead of on source code. This has the advantage of being programming language independent.

As `Pinhão` is a `LLVM` tool, the transformation rules encode each `LLVM`'s instruction. Such rules are outlined in Table 2.

Table 2: `DNA` Encoding

Transformation Rules			
Br	A	Store	K
Switch	B	Alloca	L
IndirectBr	C	Fence, AtomicRMW, AtomicCmpXchg	M
Ret, Invoke, Resume, Unreachable	D	GetElementPTR	N
Add, Sub, Mul, UDiv, SDiv, URem, SRem	E	Trunc, ZExt, SExt, UIToFP, SIToFP, PtrToInt, IntToPtr, BitCast, AddrSpaceCast	O
FAdd, FSub, FMul, FDiv, FRem	F	FPTrunc, FPExt, FPToUI, FPToSI	P
Shl, LShr, AShr, And, Or, Xor	G	ICmp, FCmp, Select, VAArg, LandingPad	Q
ExtractElement, InsertElement, SuffleVector	H	PHI	R
ExtractValue, InsertValue	I	Call	S
Load	J	Others	X

The transformation rules group instructions into different genes. As a result, `Pinhão` can identify which instruction group dominates the hot function, and use these insights for exploring potential heuristics. Appendix A presents an example of using the transformation rules.

2.3 `DNA` Sequence Aligner

Finding an effective sequence for an unseen program is based on similarity among programs. Our premise is that similar programs react approximately equal when they are compiled using the same sequence. In this manner, we need a method to find similar programs.

We determine a similar program aligning its `DNA` representation with previously-generated `DNA`s, in fact, the `DNA`s of hot functions. For this purpose, `Pinhão` uses the algorithm proposed in [Needleman and Wunsch, 1970].

Needleman and Wunsch proposed 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. Given two amino-acid sequences, A and B, Needleman-Wunsch Algorithm performs two steps:

1. Create the similarity matrix MAT; and
2. Find the maximum match.

The maximum match can be determined by a two-dimensional array, where two amino-acid sequences, A and B, are compared. Each amino-acid sequence is numbered from 1 to N, where A_j is the jth element of the sequence A and B_i is the ith element of sequence B, with A_i representing the columns and B_i the rows of the two-dimensional matrix. Then, considering the matrix MAT, MAT_{ij} represents the pair combination of A_j and B_i.

To ensure that the sequence don't have permutations of elements, a pair combination MAT_{ij} is a part of a pathway containing MAT_{mn} if and only if their indexes are $m > i$, $n > j$ or $m < i$, $n < j$. Thus, any pathway can be represented by a number of pair permutations MAT_{ab} to MAT_{yz}, where $a >= 1$, $b >= 1$, and the subsequent indexes of the cells of MAT are larger than the indexes of the previous cells and smaller than the number of elements in the respective sequences A and B. A pathway begins at a cell in the first column or first row of MAT, where the index of i and j needs to be incremented by one and the other by one or more, leading to the next cell in the pathway. Repeating this process until their limiting values creates a pathway where every partial or unnecessary pathway will be contained in at least one necessary pathway.

As a result of this process, the maximum match returns a score which indicates the similarity between the amino-acid sequences A and B.

Therefore, using Needleman-Wunsch Algorithm, `Pinhão` scores (and ranks) past experiences aligning the `DNA` of the unseen program (its hot function) with each `DNA` from the database.

2.4 Sequence Extractor

As stated before, `Pinhão` explores sequences taken from previously compiled programs, which react approximately equal when they are compiled using the same sequence.

Based on this assumption, we could conclude that the good strategy is to evaluate the best previously-generated sequence used by the most similar program. This is true if and only if we ensure that the best sequence is safe. In fact, we can not ensure that such sequence is safe[5]. Since some flags (optimizations) are unsafe, meaning that such flags can generate problems in specific programs. As a result, `Pinhão` evaluates N previously-generated sequences, in order to ensure that a safe sequence will always be returned.

After evaluating N sequences and finding the best one, which fits the characteristics of the unseen program, `Pinhão` returns the best target code or invokes `IC`.

2.5 Iterative Compiler

Invoking the `IC` is an optional step. Thus, `Pinhão` can be tunned to use this step or not. However, if the `IC` is enabled, it will be invoked if and only if the `Target Code Analyzer` indicates that the performance of the target code is not better than a threshold.

[5] A unsafe sequence will crash the compiler.

The `IC` is a genetic algorithm (`GA`), which consists in randomly generating an initial population that will be evolved in an iterative process. Such process involves choosing parents, applying genetic operators, evaluating new individuals, and finally a reinsertion operation deciding which individuals will compose the new generation. This iterative process is performed until a stopping criterion is reached.

The first generation is composed of individuals that are generated by a uniform sampling of the optimization space. Evolving a population includes the application of two genetic operators: crossover, and mutation. The former can be applied to individuals of different lengths, resulting in a new individual whose length is the average of its parents. The latter can perform four different operations, as follows:

1. insert a new optimization into a random point;
2. remove an optimization from a random point;
3. exchange two optimizations from random points; or
4. change one optimization in a random point.

Both operators have the same probability of occurrence, besides only one mutation is applied over the individual selected to be transformed. This iterative process uses a tournament strategy and elitism that maintains the best individual in the next generation. The strategy used by the `IC` is similar to the strategy proposed in [Purini and Jain, 2013] and [Martins et al., 2016].

2.6 Updating the Database

The final step is to update the database with new knowledge, in order to learn from new compilations. This means that `Pinhão` updates the database of previously-generated sequences, with information that indicates which `DNA` should be compiled using a specific sequence.

3 A Database of Previously-generated Sequences

As `Pinhão` relies on previously-generated sequences, it is necessary to construct in advance such sequences.

The database stores a pair <`DNA`, sequence> for different training programs. The `DNA` represents the program's hottest function, and the sequence is an effective sequence.

This database can be constructed in a process from factory. Thus, at the factory, an engine collects pieces of information about a set of training programs and reduces the optimization search space in order to provide a small database, which can be handled in an easy and fast way.

Training Programs Training programs are composed of programs took from `LLVM`'s test-suite [LLVM Team, 2016], and The Computer Language Benchmarks Game [Bechmarks Game Team, 2016]. These are programs composed of a single source code and have short runtime. Table 3 shows the training programs.

Table 3: Training Programs

LLVM's test-suite					
ackermann	ary3	bubblesort	chomp	dry	dt
fannkuch	fbench	ffbench	fib2	fldry	flops
flops-1	flops-2	flops-3	flops-4	flops-5	flops-6
flops-7	flops-8	fp-conver	hash	heapsort	himenobtxpa
huffbench	intmm	lists	lpbench	mandel	mandel-2
matrix	methcall	misr	n-body	nsieve-bits	oourafft
oscar	partialsums	perlin	perm	pi	puzzle
puzzle-stanford	queens	queens-mcgill	quicksort	random	realmm
recursive	reedsolomon	richards_benchmark	salsa20	sieve	spectral-norm
strcat	towers	treesort	whetstone		
The Computer Language Benchmarks Game					
binary-tree	fasta-redux	pidigits	regex-dna	fasta	mandelbrot

Optimizations Table 4 presents the optimizations which can compose a sequence.

Reducing the Search Space To reduce the search space and to find a good compiler optimization sequence for each training program, we use the `IC` described in Section 2.5.

4 Experimental Setup and Methodology

This section describes the experimental setup and the steps taken to ensure measurement accuracy. It also presents the methodology used in the experiments.

Platform The experiments were conducted on different hardware configurations, to evaluate `Pinhão`'s performance on different architectures. The hardware configurations are:

- `Hardware Configuration 1`: a machine with an Intel processor Core I7-3770 3.40GHz, and 8 GB of RAM. The operating system was Ubuntu 15.10, with kernel 4.2.0-30-generic.
- `Hardware Configuration 2`: a machine with an Intel processor Core I7-3820 3.60GHz, and 32 GB of RAM. The operating system was Ubuntu 15.10, with kernel 4.2.0-18-generic.

Table 4: Optimization Space

Optimizations				
aa-eval	adce	add-discriminators	alignment-from-assumptions	
alloca-hoisting	always-inline	argpromotion	assumption-cache-tracker	
atomic-expand	barrier	basicaa	basiccg	bb-vectorize
bdce	block-freq	bounds-checking	branch-prob	break-crit-edges
cfl-aa	codegenprepare	consthoist	constmerge	constprop
correlated-propagation		cost-model	count-aa	da
dce	deadargelim	deadarghaX0r	delinearize	die
divergence	domfrontier	domtree	dse	dwarfehprepare
early-cse	elim-avail-extern	flattencfg	float2int	functionattrs
globaldce	globalopt	globalsmodref-aa	gvn	indvars
inline	inline-cost	instcombine	instcount	instnamer
instsimplify	intervals	ipconstprop	ipsccp	irce
iv-users	jump-threading	lazy-value-info	lcssa	libcall-aa
licm	lint	load-combine	loop-accesses	loop-deletion
loop-distribute	loop-extract	loop-extract-single	loop-idiom	loop-instsimplify
loop-interchange	loop-reduce	loop-reroll	loop-rotate	loop-simplify
loop-unroll	loop-unswitch	loop-vectorize	loops	lower-expect
loweratomic	lowerbitsets	lowerinvoke	lowerswitch	mem2reg
memcpyopt	memdep	mergefunc	mergereturn	mldst-motion
nary-reassociate	no-aa	partial-inliner	partially-inline-libcalls	
place-backedge-safepoints-impl		place-safepoints	postdomtree	prune-eh
reassociate	reg2mem	regions	rewrite-statepoints-for-gc	
rewrite-symbols	safe-stack	sancov	scalar-evolution	scalarizer
scalarrepl	scalarrepl-ssa	sccp	scev-aa	scoped-noalias
separate-const-offset-from-gep		simplifycfg	sink	sjljehprepare
slp-vectorizer	slsr	speculative-execution	sroa	strip
strip-dead-prototypes		strip-nondebug	structurizecfg	tailcallelim
targetlibinfo	tbaa	tti	verify	

– `Hardware Configuration 3`: a machine with an Intel processor Core I5-2430 2.40GHz, and 4 GB of RAM. The operating system was Ubuntu 15.10, with kernel 4.2.0-30-generic.

– `Hardware Configuration 4`: one processor of a machine with two Intel processor Xeon E5504 2.00GHz, and 24 GB of RAM. The operating system was Ubuntu 15.10, with kernel 4.2.0-22-generic.

The experiments described from Section 5.1 to Section 5.3 were conducted on `Hardware Configuration 1`, and the experiments described in Section 5.4 were conducted on the four configurations.

Compiler Our technique was implemented as a tool of `LLVM` 3.7.0 [LLVM Team, 2016]. The choice of `LLVM` is based on the fact that it allows full control over the optimizations. This means that it is possible to enable a list of optimizations through the command line, where the position of each optimization indicates its order. Neither `GCC` nor `ICC` provide these features, thus we chose to use `LLVM`.

Pinhão's Parameters We performed experiments exploring 1, 3, 5 and 10 sequences. Using less than 10 sequences, `Pinhão.H` and `Pinhão.MH` are not able to surpass the performance of `LLVM` compiler optimization levels. In such case, these configurations achieve a speedup of 1.85 and 1.79, and only for 50% of the test programs. Based on these results, `Pinhão` explores 10 sequences. Furthermore, exploring 10 sequences ensures that `Pinhão` always finds a safe sequence.

Iterative Compiler's Parameters The crossover operator has a probability of 60% for creating a new individual. In this case, the tournament strategy ($Tour = 5$) selects the parents. The mutation operator has a probability of 40% for transforming an individual. In addition, each individual has an arbitrary initial length, which can ranges from 1 to the number of optimizations ($|Optimization\ Space|$). The Pinhão's `IC` (the optional step) runs over 10 generations and 20 individuals; however to create a database of previously-generated sequences, the `IC` runs over 100 generations and 50 individuals. Both `IC`s finish whether the standard deviation of the current fitness score is less than 0.01, or the best fitness score does not change in three consecutive generations.

Threshold We use as threshold the compiler optimization levels. This means that an effective sequence is the one whose performance surpasses the performance obtained by the best compiler optimization level.

Training Phase Cost The training phase, which builds the database, is a time-consuming phase. It took precisely 20 days. However, it is important to note that it was performed only one-time at the factory. The training phase was conducted on `Hardware Configuration 1`.

Testing Phase Cost The testing phase is a fast task, which includes extracting the `DNA` of the test program, scoring the training programs, selecting N sequences, and evolving the population (when the system invokes the `IC`). It takes less than 0.1% of the entire response time. It means that at least 99.9% of the time is spent evaluating target codes. Therefore, we can conclude that the machine learning process that `Pinhão` uses has almost no overhead when compared with the target code evaluation.

Benchmarks The experiments use two benchmarks-suites as unseen (test) programs, namely: POLYBENCH 4.1 [Louis-Noël Pouchet, 2016] with dataset Extralarge, and CBENCH [cBench, 2014] with dataset 1.

Order of Compilation As `Pinhão` updates the database, the order of compilation affects the results. Of course, we are not able to predict in which order the user will compile his/her programs. Thus, we decided to compile in alphabetic order.

Measurement The runtime is the arithmetic average of five executions. In the experiments, the machine workload was as minimal as possible. In other words, each instance was executed sequentially. In addition, the machine did not have an external interference, and the running time variance was close to zero.

Metrics The evaluation uses five metrics to analyze the results, namely:

1. `GMS`: geometric mean speedup ($Speedup = \frac{Runtime\_Level\_O0}{Runtime}$);
2. `GMI`: geometric mean improvement ($Impr = (Speedup - 1) * 100$);
3. `PPI`: percentage of programs achieving improvement over the *threshold*;
4. `GME`: geometric mean number of sequences evaluated; and
5. `GMT`: geometric mean response time in seconds, i.e., the time consumed by Pinhão including compilation and evaluation.

Other Techniques To evaluate the effectiveness of our technique, we compare `Pinhão` against the following five techniques.

1. Random algorithm (`Random10`). This technique randomly generates 10 sequences.
2. Genetic algorithm with tournament selector (`GA10`). This technique is the `GA` described in Section 2.5, running over 10 generations and 20 individuals.
3. Genetic algorithm with tournament selector (`GA100`). This technique is the `GA` described in Section 2.5, running over 100 generations and 50 individuals.
4. `Best10.PJ`. Purini and Jain proposed to evaluate only 10 good sequences, which are able to cover several classes of programs, and after returning the best target code. We evaluate the 10 good sequences described in [Purini and Jain, 2013].
5. `Best10`. This technique is similar to the previous one, except that we found the 10 good sequences perusing our database using the strategy proposed in [Purini and Jain, 2013].

5 Experimental Results

This section presents the results of the experiments that we performed. First of all, we evaluate the `Pinhão`'s performance. In a second moment, we evaluate the performance of `Pinhão` on different datasets. Next, we provide a direct comparison with other strategies. Finally, we evaluate again its performance but on different hardware configurations.

5.1 Pinhão Performance

As said before, our strategy is dependent on previous knowledge. Whenever a new program is compiled and the sequence performs better than the compiler optimization levels, this sequence is added to the database. This means that the order of compilation influences the quality of the results. As we do not know the order chosen by a user, we performed the experiments in alphabetic order, the same order that the results are shown in Figure 2, from left to right.

Figure 2(a) shows that 46 (Pinhão.H) and 42 (Pinhão.MH) programs out of 59 achieved speedup over all compiler optimization levels. This represents 77.97% and 71.19% of the total, respectively for Pinhão.H and Pinhão.MH. Both techniques shows good results in most of the cases, with some cases extrapolating by some times all compiler optimization levels. One example of that is the tiff2rgba, with a speedup of 8.90 for Pinhão.MH, against the speedup of 1.80, 1.72 and 1.74 from compiler optimization levels O1, O2 and O3, respectively.

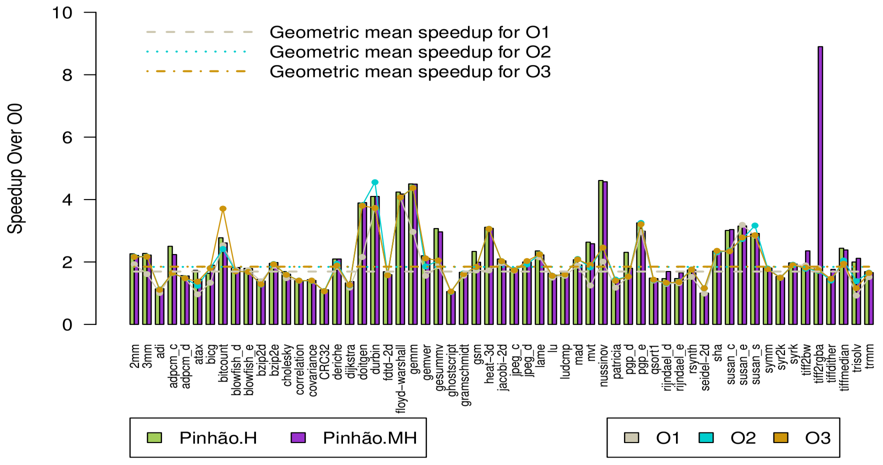
Figure 2(b) show that if after 10 tries Pinhão couldn't find an effective sequence, it applies a genetic strategy trying to find a better one. This strategy always searches for only one sequence that covers the whole program, independent of the technique adopted. With this approach, 50 (Pinhão.H+IC) and 47 (Pinhão.MH+IC) programs out of 59 reached speedup over all compiler optimization levels, representing 84.75% and 79.66% of the total, respectively for Pinhão.H+IC and Pinhão.MH+IC.

These results shows a good improvement over the original compiler optimization levels, and the confirmation that Pinhão can find an efficient sequence in the majority of the cases. This is supported by the fact that our GMS maintained a higher value than the compiler optimization levels, being 2.01, 2.02, 2.05 and 2.07 for Pinhão.H, Pinhão.H+IC, Pinhão.MH and Pinhão.MH+IC and 1.69, 1.84 and 1.85 for O1, O2 and O3, respectively.

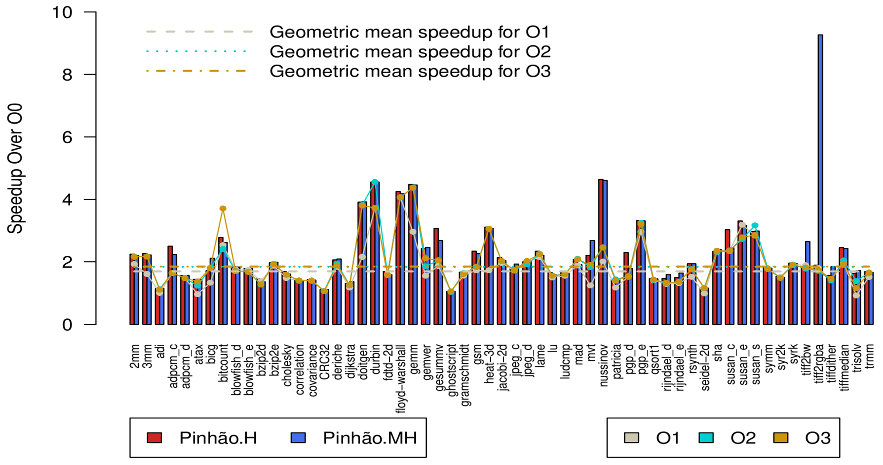
In general, our strategy always evaluates 10 sequences. However, as we mentioned before, not all sequences are safe, so there were cases where the number of sequences evaluated was less than 10. Figure 3 shows the number of sequences evaluated by Pinhão.

We observed that, in our experiments, there was only one occurrence of generating 90% of invalid sequences; besides that the GME of the Pinhão.H and Pinhão.MH was 8.17 and 6.16, respectively. Also, their standard deviation was of 1.27 sequences for Pinhão.H, and 2.66 sequences for Pinhão.HM. These data suggest that in the majority of the experiments there were at least 5 sequences evaluated.

Using the strategies coupled with the IC, we noticed that even though they evaluated up to 118 sequences, the IC strategy was only called in roughly 14% and 17% of the experiments when using the Pinhão.H and Pinhão.HM, respectively. Thus, the data suggests that our technique can find effective sequences (that



(a) `IC` Off



(b) `IC` On

Figure 2: Speedup

surpass the well-engineered ones) with up to 10 sequence evaluations, since the genetic algorithm was only used when the machine learning step did not gain performance when compared to the compiler optimization levels. It is possible to see in Figure 4 that the response time of our technique is barely the same for all 4 strategies.

Thinking about how our strategies work, we can infer that the execution

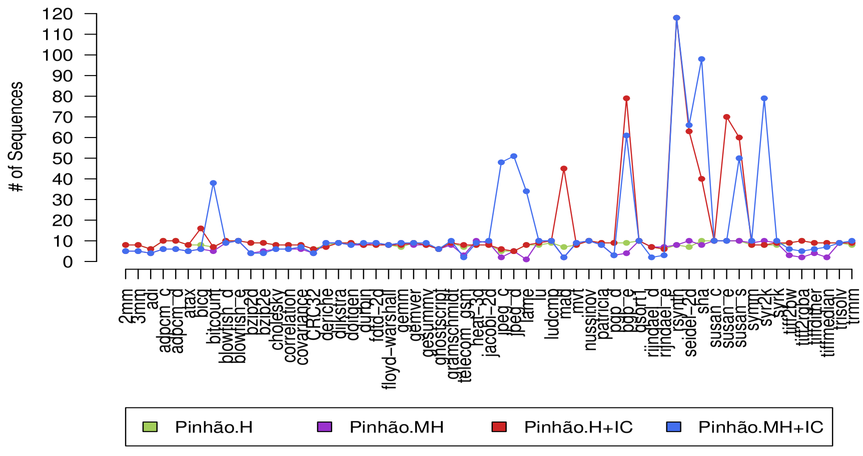


Figure 3: Number of sequences evaluated.

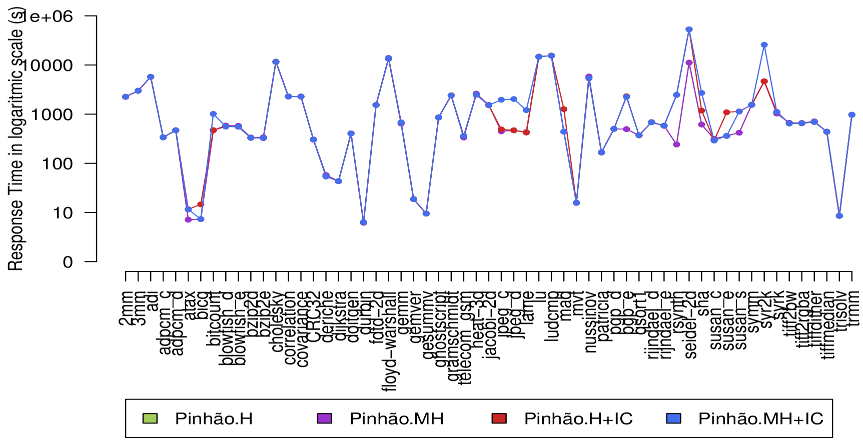


Figure 4: The time consumed by each strategy.

and testing phase, responsible for verifying if a sequence performs better than the compiler optimization levels, overwhelms the `DNA` generation and comparison. This is due to the linear characteristic of how each `DNA` is processed and compared, meaning that generating a `DNA` for multiple modules and one `DNA` for each module and comparing them is equally fast. This shows that the evaluation phase is the most consuming operation on our strategy, surpassing even the `IC`.

Moreover, we can notice that all the cases that significantly diverged from the others and consumed more time are using the `IC`. This means that sequences with lower quality affect the overall response time for `Pinhão.H+IC` and `Pinhão.MH+IC` strategies, as they applied a genetic strategy which is time-consuming.

Observing the numbers of sequences generated for each benchmark and Figure 4, a high number of sequences can imply a greater response time, although this is not valid for all cases because of the large time spent by the execution phase.

To evaluate `Pinhão`'s performance when a test program is recompiled, we performed two other experiments as follows.

1. `Pinhão.H + Pinhão.H+IC`: In this experiments `Pinhão.H+IC` uses the database generated after running `Pinhão.H`.
2. `Pinhão.MH + Pinhão.MH+IC`: In this experiments `Pinhão.MH+IC` uses the database generated after running `Pinhão.MH`.

In addition, `Pinhão` evaluates only 1 (one) sequence. Table 5 presents the results.

Table 5: Recompilation Results

Strategy	GMS	GMI	PPI	GME	GMT
Pinhão.H + Pinhão.H+IC	2.04	89.39	84.75	1.72	50.80
Pinhão.MH + Pinhão.MH+IC	2.09	92.34	81.36	2.55	68.78
O1	1.69	52.23	-	-	-
O2	1.84	70.31	-	-	-
O3	1.85	70.64	-	-	-

In these experiments, the speedups obtained a slight variation. They increase 1.99% and 0.97% for `Pinhão.H + Pinhão.H+IC` and `Pinhão.MH + Pinhão.MH+IC`, respectively. Concerning the number of programs achieving improvement, the improvement is 19.05% and 2.13% for `Pinhão.H + Pinhão.H+IC` and `Pinhão.MH + Pinhão.MH+IC`. This means that 50 and 48 programs out of 59 achieved speedup over all compiler optimization levels.

The main impact in recompiling a program occurs in the number of sequences evaluated, as well as the response time. The results show a decreasing up to 73% in the number of sequences evaluated, and a decreasing up to 90% in the response time. As the database has effective sequences, it is possible to evaluate only 1 (one) sequence. As a result, `GME` and `GMT` reflect the number of sequences evaluated when `Pinhão` invokes `IC`.

5.2 Pinhão on Different Datasets

In order to evaluate `Pinhão` performance on different open-source datasets assembled by the research community, we performed experiments with three different datasets from cBench using our strategy without resorting on the `IC`.

The performance is displayed in Figure 5 (Dataset 1), Figure 6 (Dataset 5) and Figure 7 (Dataset 10).

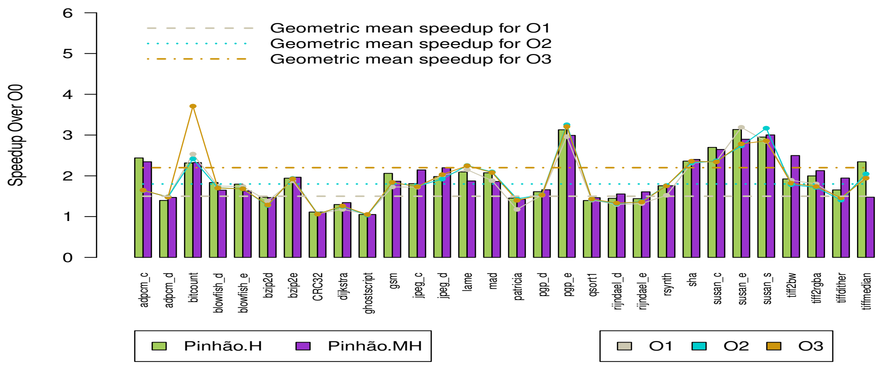


Figure 5: Results for Dataset 1

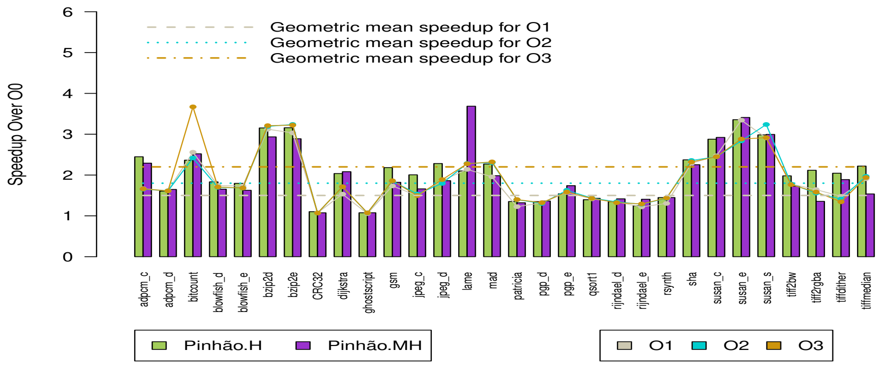


Figure 6: Results for Dataset 5

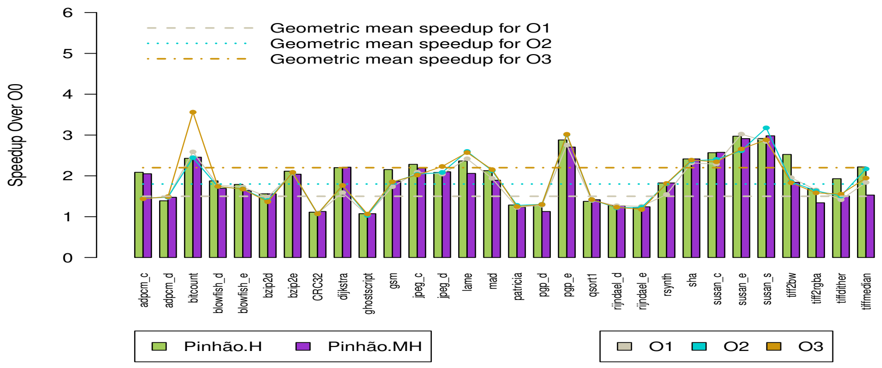


Figure 7: Results for Dataset 10

The experiments show that, overall, the strategy `Pinhão.H` was better than the compiler optimization levels and `Pinhão.MH`. The strategy was better in `73%` and `50%` of the time when compared to the compiler optimization levels and `Pinhão.MH`, respectively. However, the average improvement over the compiler sequence was between `5%` and `8%`, reaching up to `52%` faster than the well-engineered sequences.

From the results, we can infer that `Pinhão.MH` performance is highly damaged by the variations on the dataset (at least more than `Pinhão.H`). While the performance of the latter also varies - with the standard deviation of `0.068` of the average speedup between the datasets runs - it yielded a higher speedup on the majority of experiments. It is possible to verify that as it was better than the compiler optimization levels sequence on `72%` of the programs evaluated, `MH` achieved an average of `59%` among the three datasets.

5.3 Pinhão Versus Other Techniques

In Table 6 we show the results of the experiments for `Pinhão` and the other techniques.

We can notice that the geometric mean values of all techniques reveal a slight significant change in values, with the greatest geometric mean being of `Pinhão.MH+IC`, with 2.07 of speedup. The others techniques, with exception of `GA100` with 2.04 of speedup, got a better performance than the compiler optimization levels, but inferior to all of our strategies. All five metrics indicate that our technique was most effective than the others.

Although the speedup of `GA100` is considerably high, but not surpassing `Pinhão.MH`, the number of evaluated sequences is at least 82% higher than all others averages. This implies a larger response time, while our technique kept a

Table 6: `Pinhão` versus Other Techniques

Strategy	GMS	GMI	PPI	GME	GMT
Pinhão.H	2.01	81.15	77.97	8.17	518.80
Pinhão.MH	2.05	82.73	71.19	6.16	518.80
Pinhão.H+IC	2.02	87.18	84.75	10.62	617.19
Pinhão.MH+IC	2.07	90.27	79.66	9.60	663.99
Random.10	1.63	47.35	27.12	3.54	492.86
Best10.PJ	1.97	83.19	66.10	9.96	482.48
Best10	1.98	78.22	81.36	8.60	482.48
GA.10	1.80	65.90	47.46	44.62	2597.29
GA.100	2.04	85.99	71.19	242.55	15878.26
O1	1.69	52.23	-	-	-
O2	1.84	70.31	-	-	-
O3	1.85	70.64	-	-	-

comfortable amount of sequences even when using the `IC`, maintained a significantly higher speedup and improvement than all others strategies. Our response time is similar to the strategies `Random.10`, `Best10.PJ` and `Best10`, but as said before, we still yield more speedup, making the lost time worth.

Thus, from the results we can deduce that our techniques maintained a great speedup performance on all benchmarks with a reasonable response time, surpassing the well-engineered compiler optimization levels and others techniques. Despite the fact that our `IC` is time-consuming, the improvement brought by it is valuable, helping us create higher quality sequences on cases that `Pinhão.H` and `Pinhão.MH` couldn't exceed the compiler optimization levels.

5.4 Pinhão on Different Hardware Configurations

Table 7 shows the performance obtained by `Pinhão`, `Best10.PJ`, `Best10` and the compiler optimization levels on four different hardware configurations.

It is important to note that there clearly was consistency across the machines. The experiments show that, overall, `Pinhão` is better than `Best10.PJ` and `Best10`, analyzing several metrics. In general, `Pinhão` surpasses these techniques in up to 15.79%, 17.82%, 84.45% and 58.21%, analyzing `GMS`, `GMI`, `PPI` and `GME`, respectively. Analyzing the compiler optimization levels indicates a better scenario. This means that `Pinhão` always surpasses the compiler optimization levels' performance; on the other hand, this is not true on `Best10.PJ` nor `Best10`.

`Pinhão` is the best approach, since it employs a program-dependent strategy to select sequences. It does not occur on other approaches evaluated. Of course, there is an increase in response time when we evaluate several sequences; however this provides better results than the compiler optimization levels.

`Pinhão` indicates that it is possible to achieve good results evaluating a few points in the search space using a simple strategy, even for the case where training and deployment stages take place on different hardware configurations. Fur-

Table 7: Performance

Strategy	Hardware Configuration 1					Hardware Configuration 2				
	GMS	GMI	PPI	GME	GMT	GMS	GMI	PPI	GME	GMT
Pinhão.H	2.01	81.15	77.97	8.17	518.80	1.92	77.22	33.90	7.87	557.32
Pinhão.MH	2.05	82.73	71.19	6.16	518.80	1.98	77.30	66.10	5.76	623.67
Pinhão.H+IC	2.02	87.18	84.75	10.62	617.19	2.01	83.78	76.27	20.58	1003.35
Pinhão.MH+IC	2.07	90.27	79.66	9.60	663.99	2.09	90.25	74.58	9.30	957.46
Best10.PJ	1.97	83.19	66.10	9.96	482.48	1.76	60.66	11.86	10.00	598.66
Best10	1.98	78.22	81.36	8.60	482.48	1.90	74.17	33.90	8.60	606.69
O1	1.69	52.23	-	-	-	1.71	54.53	-	-	-
O2	1.84	70.31	-	-	-	1.87	72.82	-	-	-
O3	1.85	70.64	-	-	-	1.88	73.96	-	-	-

Strategy	Hardware Configuration 3					Hardware Configuration 4				
	GMS	GMI	PPI	GME	GMT	GMS	GMI	PPI	GME	GMT
Pinhão.H	2.04	86.31	71.19	8.11	611.23	2.28	105.31	77.97	8.24	879.50
Pinhão.MH	1.98	78.43	52.54	5.03	581.15	2.25	104.30	72.88	6.17	924.90
Pinhão.H+IC	2.08	92.15	72.88	13.07	834.25	2.32	115.73	84.75	15.10	1229.69
Pinhão.MH+IC	2.11	93.95	72.88	10.60	772.24	2.27	109.88	81.36	13.35	1256.34
Best10.PJ	1.87	75.22	32.20	10.00	656.78	2.00	92.42	35.59	10.00	1044.17
Best10	2.00	82.21	57.67	8.64	629.41	2.29	111.16	71.19	8.67	1017.17
O1	1.74	58.55	-	-	-	1.92	71.70	-	-	-
O2	1.88	72.77	-	-	-	2.08	93.53	-	-	-
O3	1.90	76.36	-	-	-	2.08	92.98	-	-	-

thermore, a good strategy is to inspect several hot functions, besides handling the problem of discovering what optimizations should be turned on or off as a program-dependent problem.

6 Related Work

The first-generation auto-tunning systems employ iterative compilation techniques. In such systems, the test program is compiled with different sequences, and the best version is chosen. Due to the diversity of sequences and the need of compiling and running the program several times, iterative systems try to cover the search space selectively. Based on the behavior of the search, these systems can be classified into three categories: partial search; random search; or heuristic search.

Partial search systems try to explore a portion of all possible solutions [Pan and Eigenmann, 2006, Kulkarni et al., 2009, Foleiss et al., 2011]. Random or statistical systems perform the search employing statistical and randomization techniques, in order to reduce the number of sequences evaluated [Haneda et al., 2005, Shun and Fursin, 2005, Cooper et al., 2006]. Heuristic systems use random searches based on several transformations [Kulkarni et al., 2005, Che and Wang, 2006, Zhou and Lin, 2012].

In the context of iterative compilation, an interesting work was proposed in [Purini and Jain, 2013]. Although it can be classified as a first-generation auto-tunning system, it reduces the system's response time using effective sequences, which are able to cover several programs. The process of finding effective sequences is as follows. First, using random and heuristic searches the strategy creates effective sequences for several programs. After that, the strategy selects the most effective sequence for each program, and eliminates, from each sequence, the optimizations that do not contribute to the performance. Finally, a covering algorithm analyzes all sequences and extracts the best 10 sequences. As a result, this strategy evaluates only 10 sequences to find an effective sequence for a new program.

Even though, first-generation auto-tunning systems provide good results, the problem is that they require a long response time. Thus, we decided not to implement a pure first-generation auto-tunning system. In fact, `Pinhão` is a hybrid-system and posseses several characteristics that belongs to the first-generation auto-tunning systems, as well as characteristics founded into the second-generation. It is important to note that `Pinhão` can use an iterative compiler, if it is not able to find an effective sequence in previous steps. In fact, at this moment `Pinhão` can be viewed as a first-generation auto-tunning system, which employs a heuristic-based search.

The second-generation auto-tunning systems employ machine learning techniques. The goal is to project expert systems, which is able to reduce the response time needed by systems that fit the first-generation, while finding effective sequences for an unseen program. Second-generation systems create in a training stage a prediction model, based on the behavior of several training programs. Then, in a deployment (or test) stage the prediction model predicts the sequence that will be enabled to compile the unseen program [Long and O'Boyle, 2004, Agakov et al., 2006, de Lima et al., 2013].

The prediction model creates a relation between effective sequences and characteristics of programs. It requires two steps. First, it is necessary to find effective sequences for several test programs and based on these sequences to build the model. This step is performed by an iterative compilation process like a first-generation auto-tunning system. Second, it is necessary to represent a program as a feature vector. To model a program as a feature vector, several works use different program's characteristics, such as: characteristics that describe the loop and array structure of the program [Long and O'Boyle, 2004], performance counters [Cavazos et al., 2007, de Lima et al., 2013], control-flow graphs [Park et al., 2012], compilation data [Queiroz Junior and da Silva, 2015], numerical features [Namolaru et al., 2010, Tartara and Reghizzi, 2013], or a symbolic representation, similar to a `DNA` [Martins et al., 2014]. After these two steps, it is possible to relate effective sequences to feature vectors, and so building the prediction model.

The deployment stage has been implemented using different strategies, such as: instance-based learning [Long and O'Boyle, 2004], case-based reasoning [de Lima et al., 2013, Queiroz Junior and da Silva, 2015], or logistic regression [Cavazos et al., 2007]. Such strategies infer what optimizations should be enabled [Cavazos et al., 2007], or what sequence should be used [de Lima et al., 2013, Queiroz Junior and da Silva, 2015].

Although `Pinhão` employs a hybrid approach, it is primarily a machine learning technique. `Pinhão` models a program using a symbolic representation, similar to a `DNA`, and based on this representation `Pinhão` infers what sequence should be used and not what optimizations should be enabled or disabled. This process is similar to a case-based reasoning strategy.

As stated before, `Pinhão` explores the premise that hot functions are the portions at which compiler optimizations will provide the greatest benefit. So that, the auto-tunning system is guided by such functions. The works [Long and O'Boyle, 2004] and [Hoste et al., 2010] are close to `Pinhão`, concerning to the use of hot functions. However, while `Pinhão` extracts hot functions from `C` source code, these works explore a Java Virtual Machine [Alpern et al., 2000, M. Paleczny and C. Vick and C. Click, 2001]. This means that these works implicitly focus on hot functions, due to modern Java Virtual Machines employ different compilations plans on hot functions.

The third-generation auto-tunning systems employ a long-term machine learning technique. Such system tries to learn from every compilation, without employing a training stage.

The work [Tartara and Reghizzi, 2013] demonstrated that is possible to eliminate the training stage, using a long-term learning. The strategy performs two tasks. First, it extracts the characteristics of the test program. Second, a genetic algorithm creates heuristics inferring which optimizations should be enabled during target code generation. This process creates knowledge that is used in new compilations.

`Pinhão` and Tartara's and Crespi's work differ at least in two points. First, they characterize programs using different features. The former uses a symbolic representation, similar to a `DNA`. The latter uses the numeric features proposed in [Namolaru et al., 2010]. Second, which is the most important one, `Pinhão` fits into the second-generation, while Tartara's and Crespi's work fits into the third-generation of auto-tunning systems.

7 Concluding Remarks

Finding an effective compiler optimization sequence is a program-dependent problem. For that, a good strategy is to inspect the characteristics of the program, and based on these characteristics to explore the search space looking for effective sequences. In addition, a considerable amount of runtime is spent in a small portion of code. Therefore, the ideal features to consider is that extracted from hot functions.

In this paper, we proposed `Pinhão` an auto-tunning system for compiler optimizations, which is guided by hot functions. This means that `Pinhão` finds the compiler optimization sequence that will be enabled during target code generation, inspecting hot functions.

`Pinhão` is a fast auto-tunning system, which finds effective sequences and outperforms traditional iterative compilation techniques.

References

[Agakov et al., 2006] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. (2006). Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA. IEEE Computer Society.

[Alpern et al., 2000] Alpern, B., Attanasio, C. R., Barton, J. J., Burke, M. G., Cheng, P., Choi, J.-D., Cocchi, A., Fink, S. J., Grove, D., Hind, M., Hummel, S. F., Lieber, D., Litvinov, V., Mergen, M. F., Ngo, T., Russell, J. R., Sarkar, V., Serrano, M. J., Shepherd, J. C., Smith, S. E., Sreedhar, V. C., Srinivasan, H., and Whaley, J. (2000). The Jalapeño Virtual Machine. *IBM System Journal*, 39(1):211–238.

[Ball and Larus, 1993] Ball, T. and Larus, J. R. (1993). Branch Prediction for Free. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 300–313, New York, NY, USA. ACM.

[Bechmarks Game Team, 2016] Bechmarks Game Team (2016). The Computer Language Benchmarks Game. http://http://benchmarksgame.alioth.debian.org/ Access: January, 20 - 2016.

[Cavazos et al., 2007] Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

[cBench, 2014] cBench (2014). The Collective Benchmarks. http://ctuning.org/wiki/index.php/CTools:CBench. Access: January, 20 - 2016.

[Che and Wang, 2006] Che, Y. and Wang, Z. (2006). A Lightweight Iterative Compilation Approach for Optimization Parameter Selection. In *First International Multi-Symposiums on Computer and Computational Sciences*, volume 1, pages 318–325, Washington, DC, USA. IEEE Computer Society.

[Cooper and Torczon, 2011] Cooper, K. and Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann, USA, 2nd edition.

[Cooper et al., 2006] Cooper, K. D., Grosul, A., Harvey, T. J., Reeves, S., Subramanian, D., Torczon, L., and Waterman, T. (2006). Exploring the Structure of the Space of Compilation Sequences Using Randomized Search Algorithms. *Journal of Supercomputing*, 36(2):135–151.

[de Lima et al., 2013] de Lima, E. D., de Souza Xavier, T. C., da Silva, A. F., and Ruiz, L. B. (2013). Compiling for Performance and Power Efficiency. In *Proceedings of the International Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 142–149.

[Foleiss et al., 2011] Foleiss, J. H., da Silva, A. F., and Ruiz, L. B. (2011). An Experimental Evaluation of Compiler Optimizations on Code Size. In *Proceedings of the Brazilian Symposium on Programming Languages*, pages 1–15, São Paulo, São Paulo, Brazil.

[Haneda et al., 2005] Haneda, M., Knijnenburg, P. M. W., and Wijshoff, H. A. G. (2005). Generating New General Compiler Optimization Settings. In *Proceedings of* *the Annual International Conference on Supercomputing*, pages 161–168, New York, NY, USA. ACM.

[Hoste et al., 2010] Hoste, K., Georges, A., and Eeckhout, L. (2010). Automated Just-in-time Compiler Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 62–72, New York, NY, USA. ACM.

[Kulkarni et al., 2005] Kulkarni, P. A., Hines, S. R., Whalley, D. B., Hiser, J. D., Davidson, J. W., and Jones, D. L. (2005). Fast and Efficient Searches for Effective Optimization-Phase Sequences. *ACM Transactions on Architecture and Code Optimization*, 2(2):165–198.

[Kulkarni et al., 2009] Kulkarni, P. A., Whalley, D. B., Tyson, G. S., and Davidson, J. W. (2009). Practical Exhaustive Optimization Phase Order Exploration and Evaluation. *ACM Transactions on Architecture and Code Optimization*, 6(1):1–36.

[Kulkarni and Cavazos, 2012] Kulkarni, S. and Cavazos, J. (2012). Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 147–162, New York, NY, USA. ACM.

[LLVM Team, 2016] LLVM Team (2016). The LLVM Compiler Infrastructure. http://llvm.org. Access: January, 20 - 2016.

[Long and O'Boyle, 2004] Long, S. and O'Boyle, M. (2004). Adaptive Java Optimisation Using Instance-based Learning. In *Proceedings of the International Conference on Supercomputing*, pages 237–246, New York, NY, USA. ACM.

[Louis-Noël Pouchet, 2016] Louis-Noël Pouchet (2016). The Polyhedral Benchmark Suite. http://www.cs.ucla.edu/pouchet/software/polybench/. Acess: January, 20 - 2016.

[M. Paleczny and C. Vick and C. Click, 2001] M. Paleczny and C. Vick and C. Click (2001). The Java Hotspot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12, Monterey, CA, USA.

[Martins et al., 2014] Martins, L. G., Nobre, R., Delbem, A. C., Marques, E., and Cardoso, J. a. M. (2014). Exploration of Compiler Optimization Sequences Using Clustering-based Selection. *SIGPLAN Notices*, 49(5):63–72.

[Martins et al., 2016] Martins, L. G. A., Nobre, R., Cardoso, J. a. M. P., Delbem, A. C. B., and Marques, E. (2016). Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code Optimization*, 13(1):8:1–8:28.

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

[Namolaru et al., 2010] Namolaru, M., Cohen, A., Fursin, G., Zaks, A., and Freund, A. (2010). Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 197–206, New York, NY, USA. ACM.

[Needleman and Wunsch, 1970] Needleman, S. B. and Wunsch, C. D. (1970). A general Method Applicable to The Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453.

[Pan and Eigenmann, 2006] Pan, Z. and Eigenmann, R. (2006). Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

[Park et al., 2012] Park, E., Cavazos, J., and Alvarez, M. A. (2012). Using Graph-based Program Characterization for Predictive Modeling. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–206, New York, NY, USA. ACM.

[Park et al., 2011] Park, E., Kulkarni, S., and Cavazos, J. (2011). An Evaluation of Different Modeling Techniques for Iterative Compilation. In *Proceedings of the In-*

ternational Conference on Compilers, Architectures and Synthesis for Embedded Systems, pages 65–74, New York, NY, USA. ACM.

[Purini and Jain, 2013] Purini, S. and Jain, L. (2013). Finding Good Optimization Sequences Covering Program Space. *ACM Transactions on Architecture and Code Optimization*, 9(4):1–23.

[Queiroz Junior and da Silva, 2015] Queiroz Junior, N. L. and da Silva, A. F. (2015). Finding Good Compiler Optimization Sets - A Case-based Reasoning Approach. In *Proceedings of the International Conference on Enterprise Information Systems*, pages 504–515, Spain.

[Sanches and Cardoso, 2010] Sanches, A. and Cardoso, J. M. P. (2010). On identifying patterns in code repositories to assist the generation of hardware templates. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 267–270, Washington, DC, USA. IEEE Computer Society.

[Shun and Fursin, 2005] Shun, L. and Fursin, G. (2005). A Heuristic Search Algorithm Based on Unified Transformation Framework. In *Proceedings of the International Conference Workshops on Parallel Processing*, pages 137–144, Oslo, Norway. IEEE Computer Society.

[Tartara and Reghizzi, 2013] Tartara, M. and Reghizzi, S. C. (2013). Continuous Learning of Compiler Heuristics. *ACM Transaction on Architecture an Code Optimization*, 9(4):46:1–46:25.

[Wu and Larus, 1994] Wu, Y. and Larus, J. R. (1994). Static Branch Frequency and Program Profile Analysis. In *Proceedings of the International Symposium on Microarchitecture*, pages 1–11, New York, NY, USA. ACM.

[Zhou and Lin, 2012] Zhou, Y. Q. and Lin, N. W. (2012). A Study on Optimizing Execution Time and Code Size in Iterative Compilation. In *Proceedings of the International Conference on Innovations in Bio-Inspired Computing and Applications*, pages 104–109.

A An Example

This appendix provides an example of extracting a hot function, and after transforming this function into a `DNA`. The program in this example computes `PI` by probability, and was taken from `LLVM`'s test suite [LLVM Team, 2016]. This program is a training program used by our system.

As stated in Section 3, our system relies on previously-generated sequences. So that, a database stores a pair <`DNA`, sequence> for different training programs. To store such pair, our proposed system performs the following steps:

– Transform the source code (`C` language) into `LLVM`'s instructions;

– Find the cost of each function represented in `LLVM`'s instruction;

– Extract the hottest function (`LLVM`'s instruction); and

– Transform the hottest function into a `DNA`.

Subsection A.1 presents the source code of the training program. Subsection A.2 presents the training program in `LLVM's` instruction. Subsection A.3 presents the cost of each training program's function. Subsection A.4 presents the training program's hottest function as a `DNA`.

A.1 Program PI in C Language

```
#include <stdio.h>

void myadd(float *sum,float *addend) {
      *sum = *sum + *addend;
}

int main(int argc, char *argv[]) {
   float ztot, yran, ymult, ymod, x, y, z, pi, prod;
   long int low, ixran, itot, j, iprod;

   printf("Starting PI...\n");
   ztot = 0.0;
   low = 1;
   ixran = 1907;
   yran = 5813.0;
   ymult = 1307.0;
   ymod = 5471.0;
#ifdef SMALL_PROBLEM_SIZE
   itot = 4000000;
#else
   itot = 40000000;
#endif

   for(j=1; j<=itot; j++) {
        iprod = 27611 * ixran;
        ixran = iprod - 74383*(long int)(iprod/74383);
        x = (float)ixran / 74383.0;
        prod = ymult * yran;
        yran = (prod - ymod*(long int)(prod/ymod));
        y = yran / ymod;
        z = x*x + y*y;
        myadd(&ztot,&z);
        if ( z <= 1.0 ) {
          low = low + 1;
        }
   }
   printf(" x = %9.6f    y = %12.2f  low = %8d j = %7d\n",x,y,(int)low,(int)j);
   pi = 4.0 * (float)low/(float)itot;
   printf("Pi = %9.6f ztot = %12.2f itot = %8d\n",pi,ztot*0.0,(int)itot);
   return 0;
}
```

A.2 Program PI in LLVM's Instructions

```
; ModuleID = 'pi.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@.str = private unnamed_addr constant [16 x i8] c\
       "Starting PI...\0A\00", align 1
@.str.1 = private unnamed_addr constant [45 x i8] c\
       " x = %9.6f    y = %12.2f  low = %8d j = %7d\0A\00", align 1
@.str.2 = private unnamed_addr constant [37 x i8] c\
       "Pi = %9.6f ztot = %12.2f itot = %8d\0A\00", align 1

; Function Attrs: nounwind uwtable
define void @myadd(float* %sum, float* %addend) #0 {
  %1 = alloca float*, align 8
  %2 = alloca float*, align 8
  store float* %sum, float** %1, align 8
  store float* %addend, float** %2, align 8
  %3 = load float*, float** %1, align 8
  %4 = load float, float* %3, align 4
  %5 = load float*, float** %2, align 8
  %6 = load float, float* %5, align 4
  %7 = fadd float %4, %6
  %8 = load float*, float** %1, align 8
  store float %7, float* %8, align 4
  ret void
}

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  %ztot = alloca float, align 4
  %yran = alloca float, align 4
  %ymult = alloca float, align 4
  %ymod = alloca float, align 4
  %x = alloca float, align 4
  %y = alloca float, align 4
  %z = alloca float, align 4
  %pi = alloca float, align 4
  %prod = alloca float, align 4
  %low = alloca i64, align 8
  %ixran = alloca i64, align 8
  %itot = alloca i64, align 8
  %j = alloca i64, align 8
  %iprod = alloca i64, align 8
  store i32 0, i32* %1, align 4
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = call i32 (i8*, ...) @printf(i8* getelementptr\
         inbounds ([16 x i8], [16 x i8]* @.str, i32 0, i32 0))
  store float 0.000000e+00, float* %ztot, align 4
  store i64 1, i64* %low, align 8
  store i64 1907, i64* %ixran, align 8
  store float 5.813000e+03, float* %yran, align 4
  store float 1.307000e+03, float* %ymult, align 4
  store float 5.471000e+03, float* %ymod, align 4
  store i64 40000000, i64* %itot, align 8
  store i64 1, i64* %j, align 8
  br label %5

; <label>:5                                       ; preds = %51, %0
  %6 = load i64, i64* %j, align 8
  %7 = load i64, i64* %itot, align 8
  %8 = icmp sle i64 %6, %7
  br i1 %8, label %9, label %54

; <label>:9                                       ; preds = %5
  %10 = load i64, i64* %ixran, align 8
  %11 = mul nsw i64 27611, %10
  store i64 %11, i64* %iprod, align 8
  %12 = load i64, i64* %iprod, align 8
  %13 = load i64, i64* %iprod, align 8
  %14 = sdiv i64 %13, 74383
  %15 = mul nsw i64 74383, %14
  %16 = sub nsw i64 %12, %15
  store i64 %16, i64* %ixran, align 8
  %17 = load i64, i64* %ixran, align 8
  %18 = sitofp i64 %17 to float
  %19 = fpext float %18 to double
  %20 = fdiv double %19, 7.438300e+04
  %21 = fptrunc double %20 to float
  store float %21, float* %x, align 4
  %22 = load float, float* %ymult, align 4
  %23 = load float, float* %yran, align 4
  %24 = fmul float %22, %23
  store float %24, float* %prod, align 4
  %25 = load float, float* %prod, align 4
  %26 = load float, float* %ymod, align 4
  %27 = load float, float* %prod, align 4
  %28 = load float, float* %ymod, align 4
  %29 = fdiv float %27, %28
  %30 = fptosi float %29 to i64
  %31 = sitofp i64 %30 to float
  %32 = fmul float %26, %31
  %33 = fsub float %25, %32
  store float %33, float* %yran, align 4
  %34 = load float, float* %yran, align 4
  %35 = load float, float* %ymod, align 4
  %36 = fdiv float %34, %35
  store float %36, float* %y, align 4
  %37 = load float, float* %x, align 4
  %38 = load float, float* %x, align 4
  %39 = fmul float %37, %38
  %40 = load float, float* %y, align 4
  %41 = load float, float* %y, align 4
  %42 = fmul float %40, %41
  %43 = fadd float %39, %42
  store float %43, float* %z, align 4
  call void @myadd(float* %ztot, float* %z)
  %44 = load float, float* %z, align 4
  %45 = fpext float %44 to double
  %46 = fcmp ole double %45, 1.000000e+00
  br i1 %46, label %47, label %50

; <label>:47                                      ; preds = %9
  %48 = load i64, i64* %low, align 8
  %49 = add nsw i64 %48, 1
  store i64 %49, i64* %low, align 8
  br label %50

; <label>:50                                      ; preds = %47, %9
  br label %51

; <label>:51                                      ; preds = %50
  %52 = load i64, i64* %j, align 8
  %53 = add nsw i64 %52, 1
  store i64 %53, i64* %j, align 8
  br label %5

; <label>:54                                      ; preds = %5
  %55 = load float, float* %x, align 4
  %56 = fpext float %55 to double
  %57 = load float, float* %y, align 4
  %58 = fpext float %57 to double
  %59 = load i64, i64* %low, align 8
  %60 = trunc i64 %59 to i32
  %61 = load i64, i64* %j, align 8
  %62 = trunc i64 %61 to i32
  %63 = call i32 (i8*, ...) @printf(i8* getelementptr\
           inbounds ([45 x i8], [45 x i8]* @.str.1, i32 0, i32 0),\
           double %56, double %58, i32 %60, i32 %62)
  %64 = load i64, i64* %low, align 8
  %65 = sitofp i64 %64 to float
  %66 = fpext float %65 to double
  %67 = fmul double 4.000000e+00, %66
  %68 = load i64, i64* %itot, align 8
  %69 = sitofp i64 %68 to float
  %70 = fpext float %69 to double
  %71 = fdiv double %67, %70
  %72 = fptrunc double %71 to float
  store float %72, float* %pi, align 4
  %73 = load float, float* %pi, align 4
  %74 = fpext float %73 to double
  %75 = load float, float* %ztot, align 4
  %76 = fpext float %75 to double
  %77 = fmul double %76, 0.000000e+00
  %78 = load i64, i64* %itot, align 8
  %79 = trunc i64 %78 to i32
  %80 = call i32 (i8*, ...) @printf(i8* getelementptr\
           inbounds ([37 x i8], [37 x i8]* @.str.2, i32 0, i32 0),\
           double %74, double %77, i32 %79)
  ret i32 0
```

```
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { nounwind uwtable "disable-tail-calls"="false"\
                  "less-precise-fpmad"="false" "no-frame-pointer-elim"="true"\
                  "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false"\
                  "no-nans-fp-math"="false" "stack-protector-buffer-size"="8"\
                  "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2"\
                  "unsafe-fp-math"="false" "use-soft-float"="false" }
attributes #1 = { "disable-tail-calls"="false" "less-precise-fpmad"="false"\
                  "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"\
                  "no-infs-fp-math"="false" "no-nans-fp-math"="false"\
                  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"\
                  "target-features"="+fxsr,+mmx,+sse,+sse2" "unsafe-fp-math"="false"\
                  "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.7.0 (tags/RELEASE_370/final)"}
```

A.3 Cost of `PI`'s Functions

The cost finds by `Wu-Larus Static Analysis Profiler` for each function is:

- myadd = 12.0,
- main = 1050.6954.

Based on these results we can conclude that the function main is the hottest function. As a result, the database will store the `DNA` of this function and an effective sequence for this `DNA`.

A.4 Program `PI`'s Hottest Function as a `DNA`

```
LLLLLLLLLLLLLLLLLKKKSKKKKKKKKAJJQAJEKJJEEEKOPFPKJJJJFPOFFKJJFK \
JJFJJFFKSJPQAJEKAAJEKAJPJPJOJOSJOPFJOPFPKJPJPFJOS
```