Optimizing the Order of Bytecode Handlers in Interpreters using a Genetic Algorithm

Wanhong Huang
The University of Tokyo
Japan
huang-wanhong@g.ecc.u-tokyo.ac.jp

Stefan Marr
University of Kent
United Kingdom
s.marr@kent.ac.uk

Tomoharu Ugawa
The University of Tokyo
Japan
tugawa@acm.org

ABSTRACT

Interpreter performance remains important today. Interpreters are needed in resource constrained systems, and even in systems with just-in-time compilers, they are crucial during warm up. A common form of interpreters is a bytecode interpreter, where the interpreter executes bytecode instructions one by one. Each bytecode is executed by the corresponding bytecode handler.

In this paper, we show that the order of the bytecode handlers in the interpreter source code affects the execution performance of programs on the interpreter. On the basis of this observation, we propose a genetic algorithm (GA) approach to find an approximately optimal order. In our GA approach, we find an order optimized for a specific benchmark program and a specific CPU.

We evaluated the effectiveness of our approach on various models of CPUs including x86 processors and an ARM processor. The order found using GA improved the execution speed of the program for which the order was optimized between 0.8% and 23.0% with 7.7% on average. We also assess the cross-benchmark and cross-machine performance of the GA-found order. Some orders showed good generalizability across benchmarks, speeding up all benchmark programs. However, the solutions do not generalize across different machines, indicating that they are highly specific to a microarchitecture.

CCS CONCEPTS

• **Computing methodologies** → **Genetic algorithms**; • **Software and its engineering** → **Interpreters**.

KEYWORDS

Interpreters, Genetic Algorithm, Code Layout, JavaScript, Embedded Systems

ACM Reference Format:
Wanhong Huang, Stefan Marr, and Tomoharu Ugawa. 2023. Optimizing the Order of Bytecode Handlers in Interpreters using a Genetic Algorithm. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3555776.3577712

SAC '23, March 27-March 31, 2023, Tallinn, Estonia

1 INTRODUCTION

Interpreters play an important role in programming language implementations. While performance-oriented virtual machines (VMs) use just-in-time (JIT) compilers, interpreters remain prevalent because (1) interpreters are easier to implement, (2) they work well in resource-constrained environments, (3) JIT compiling all code in large applications may cause the compilation cost to be higher than the run-time saving for rarely executed code, and finally, (4) interpreters commonly have less startup cost.

A common form of interpreters executes programs by first compiling them into an assembly-like intermediate language called bytecode. Bytecode interpreters have an *instruction dispatcher*, which decodes the bytecodes, selects the *bytecode handler* to execute the bytecode, and jumps to the handler.

Thread code [2] is a popular optimization for these bytecode interpreters. Without the threaded-code optimization, whenever a bytecode handler completes its execution, it jumps to the single dispatcher, which typically resides at the head of the outer loop. In a threaded-code interpreter, in contrast, each bytecode handler has a dispatcher and jumps directly to the next bytecode.

In this paper, we improve the performance of threaded-code interpreters by rearranging the order of the bytecode handlers in the interpreter program. As previous studies suggested [4, 15, 16, 18, 19], rearrangement of basic blocks can change performance, because processor microarchitectures improve the execution speed using the location of code, e.g., caches and branch predictors. We observed that the order of the bytecode handlers significantly impacts the execution time (see Section 2.4) using a JavaScript interpreter, eJSVM [10] (see Section 2.1).

Finding the optimal order of handlers is challenging. A simple heuristics-based solution construction approach from intuition may result in a suboptimal solution because many microarchitectural factors may affect performance. Assuming that locality of executed program code dominates the performance factors, arranging handlers in the order of execution frequency may improve performance. However, these heuristics do not always give optimal performance as we see in Section 2.4. In addition, the factors and the weight of the factors differ from one microarchitecture to another. This means, for best performance, one may need to use different interpreter binaries, each optimized for a particular microarchitecture.

We used a *genetic algorithm* (GA) to find an approximately optimal handler order. In our GA approach, a solution determines the order of handlers. The quality of a solution is measured by the average execution time of a benchmark program using the interpreter in which handlers are arranged in the order determined by the solution. Execution times are measured on the target microarchitecture.

Thus, the optimal solution reflects the characteristics of the *target microarchitecture*.

Since we use a specific program on a specific microarchitecture to select the optimal solution, the solution may be specific not only to the microarchitecture but also the program. In other words, a solution produced by the GA, which we call *GA solution*, may be optimized for the *target program* and the target microarchitecture. For the embedded systems that we target, this is practical because we can know the program installed in the system together with the interpreter. Some systems take this approach even further and specialize the interpreter for a specific program [7, 10]. However outside of embedded systems, generalizable solutions are desirable; one would ideally want a single solution to benefit all programs.

We evaluated GA solutions on three Intel x86 processors and one ARM processor, and we confirmed that the GA solutions improved the performance for the target benchmark programs. The speedups were between 0.8% and 23.0% with 7.7% speedup on average. We also evaluated the generalizability of GA solutions and found that a GA solution optimized for a benchmark program is likely to speed up executions of other benchmarks on the same target microarchitecture. We observed 4.6% speedup on average when we used solutions optimized for different benchmarks on x86 processors.

2 BACKGROUND

2.1 eJS

The eJSVM is a JavaScript VM designed for embedded devices and uses a register-based bytecode interpreter. The interpreter optimizes bytecode dispatch using threaded code. The eJS compiler compiles JavaScript programs to bytecode before execution. The bytecode set consists of 58 bytecodes that include constant loading, arithmetic, control, object, and closure related bytecodes.

The eJSVM optimizes the bytecode using constant propagation, copy propagation, a limited form of common subexpression elimination, and dead code elimination. It also implements a register allocator to reduce the number of required software registers. All optimizations together decrease the bytecode size of programs by 26.8% percent on average.

A unique feature of the eJSVM is that the VM construction framework [10] generates an optimized eJSVM for a particular JavaScript program based on profiling information obtained from the program's execution. In embedded systems, developers typically deploy the JavaScript program together with the VM. Therefore, the VM can be optimized for the specific JavaScript program. Using profiling information, eJSVM can be specialized to only support the needed datatypes in a bytecode handler and thus generate a simplified and more optimal bytecode handler [8, 10]. It can also construct superinstructions from frequently used pairs of constant loading bytecode instructions and arithmetic bytecode instructions.

However, to reduce the number of possible bytecode handler orders and enable cross-benchmark comparisons, we disabled superinstructions and handler specializations in this paper. Thus, we allowed all datatypes and did not construct superinstructions.

2.2 Threaded-code Interpreter

A simple way to implement a bytecode interpreter is to use a big switch-case statement surrounded by a loop, called the *interpreter loop*. The pseudocode of this kind of interpreter is shown in Figure 1. At the head of the interpreter loop, it fetches the next bytecode and decodes it. Then, the switch-case statement dispatches to the handler corresponding to the bytecode. When an execution of any bytecode handler completes, the handler jumps back to the head of the interpreter loop to handle the next bytecode.

```
interpreter(VM_Context *ctx) {
  pc = 0;
  while (true) {
    switch (ctx->bytecode[pc++]) {
    case BYTECODE_1:
      handler for bytecode_1
      break;
    case BYTECODE_2:
      handler for bytecode_2
      break;
    ...
    case BYTECODE_n:
      handler for bytecode_n
      break;
    }
  }
}
```

Figure 1: Interpreter without threaded-code optimization.

```
#define NEXT_INSN goto *ctx->bytecode_handler[pc++]
interpreter(VM_Context *ctx) {
  pc = 0;
  NEXT_INSN;
case BYTECODE_1:
  handler for bytecode_1
  NEXT_INSN;
case BYTECODE_2:
  handler for bytecode_2
  NEXT_INSN;
...
case BYTECODE_n:
  handler for bytecode_n
  NEXT_INSN;
}
```

Figure 2: Thread-code interpreter.

In an interpreter with the *direct threaded code* optimization, the bytecode handlers do not jump back to the head of the interpreter loop. Instead, each bytecode handler dispatches to the handler for the next bytecode at the end. The pseudocode of a direct threaded interpreter is shown in Figure 2. The bytecode numbers in the bytecode program of a function are converted to the addresses of the corresponding handlers in advance. At the end of each handler, it fetches the address of the next bytecode and jumps to the address in `NEXT_INSN`.

By eliminating jumps to the head of the interpreter loop, threaded code avoids one jump per bytecode and reduces branch prediction misses on common microarchitectures [3]. This is because bytecode sequences typically show repeated patterns, and programs rarely exhibit all possible bytecode sequences. Thus, the branch predictor can learn the possible branch targets, i.e., the most likely successor bytecode for each bytecode handler separately based on the jump machine instruction at the end of each handler.

2.3 Opportunity of Handler Reordering Optimization

If a bytecode X is very likely to be executed consecutively after a bytecode Y, arranging the bytecode handler of Y immediately

after that of X will improve performance. In this arrangement, the execution of machine code for the sequence of bytecodes $X;Y$ falls though from the handler of X to that of Y, which may make the combination fit better into the instruction cache and help the prefetcher or other facilities of the microarchitecture.

In addition to code locality, reordering of bytecode handlers may affect performance in other ways. Reduction of the branch prediction misses we mention above is but one example. Many other factors may affect performance, e.g., as discussed in the work on Stabilizer [6] and CodeShaker [21].

In its standard configuration, eJSVM arranges the bytecode handlers in the order of bytecode numbers. Numbers are assigned so that the bytecodes in the same category have closer numbers. For example, arithmetic bytecodes have consecutive numbers. Therefore, the order of eJSVM's bytecode handlers may not be optimal in terms of performance. Hence, we expect that reordering the bytecode handlers improves performance.

2.4 Preliminary Evaluation

To verify our assumption that reordering the handler can change the execution speed of the program, we compared two VMs with different orders of handlers. One is the *bytecode-number order*, which is the order of the original eJSVM. The other is the *frequency order* of each program in which the handlers are ordered from the most to the least frequently executed. The frequencies are measured by profiling the execution of each program in advance. The frequency order is based on the assumption that better code locality improves performance. This experiment is conducted on three x86 processors and an ARM processor (see Section 4.1).

Figure 3 shows the speedups for the frequency order over the bytecode-number order for each benchmark on the Y-axis; the higher, the better. For Bounce and Permute, we observed -14.5% and -12.8% of speedups, i.e., slowdown, for x86A, but consistently better results on RPi's ARM processor.

As shown in the figure, the order of the bytecode handlers affected the execution speed. Simply changing the order resulted in speedups from +8.5% to -15.6%, depending on the program and the processor. This result suggests an opportunity of the handler reordering optimization.

However, the result suggested that not only code locality but also other factors affected performance. For three out of four types of hardware, performance was improved on average. Therefore, we think code locality is one of the sources of performance improvements. However, the execution was slowed down for some cases. This implies that other factors than code locality affected performance. Furthermore, how the frequency order affected performance differed from processor to processor. This motivated us to find better orders by using GA.

3 GENETIC ALGORITHM

Our goal is to find the handler order that improves performance as much as possible, for which we use a genetic algorithm [9] (GA). GA is useful since we do not know exactly which and by how much different microarchitectural factors affect performance. Furthermore, we expect modern microarchitectures to differ widely, which makes GA, as a metaheuristic search, a good candidate.

The GA takes an initial set of candidates, identifies good ones, and produces a new set of candidates based on them. A round of this process is called a *generation*. Taking the randomly chosen *initial candidates*, GA tries to improve over them a little with each generation.

In the GA framework, each candidate solution is encoded as a sequence, called *chromosome*. New chromosomes are created by *crossover* and *mutation* operations. The crossover operation produces a new chromosome by combining parts of two *parent* chromosomes. The mutation operation produces a new chromosome by mutating a single chromosome. After generating many chromosomes in a generation, GA *selects* good chromosomes from them and uses the selected ones as the parents for the next generation.

The selection operation evaluates each chromosome. We use the benchmark execution time as a selection criterion, where a candidate is better if it has a shorter execution time. To measure the result, we build a new VM for each chromosome and run a benchmark on the VM.

3.1 Solution Representation

A solution for us is a specific order of the bytecode handlers. We will evaluate two representations for solutions in Section 4: the *path representation* and the *adjacency representation*. They are common representations used to encode the traveling salesman problem (TSP) [13] for GA. We chose these two because, we have the same constraints as the TSP: a solution must be a permutation of a sequence where each element appears exactly once.

The path representation is a straightforward representation. A chromosome is an array of bytecode numbers. With this chromosome, the bytecode handlers are arranged in the order of the bytecode numbers in the array.

With the adjacency representation, we can express that an bytecode X is likely to be followed directly by a bytecode Y. A chromosome in the adjacency representation is an array indexed by bytecode numbers. An element of the array indicates the bytecode number of the following bytecode. For a chromosome C, if $C[i] = j$, then the bytecode handlers are ordered so that the handler for the bytecode i is followed by that for the bytecode j.

A valid adjacency representation expresses a cycle comprising all bytecode handlers. To decide the first handler in the list of handlers, we added a pseudo bytecode, *entry*. The bytecode handlers are arranged in the order of the cycle, starting from the next handler to the entry.

3.2 Crossover

For the path representation, we used the partially-mapped crossover operator [13]. It chooses two random indexes and swaps the contents of two parent chromosomes for the indexes between the randomly chosen indexes. At the same time, it uses the swapped part as a partial function, f, to *repair* the sequence to make it a valid representation. The partial function f maps an element in a swapped part of a parent to the other parent's element at the same index. For an element X in the unswapped part, the same element may come from the other parent by swapping. In such a case, it replace X with $f(X)$. If $f(X)$ also comes by swapping, it replaces

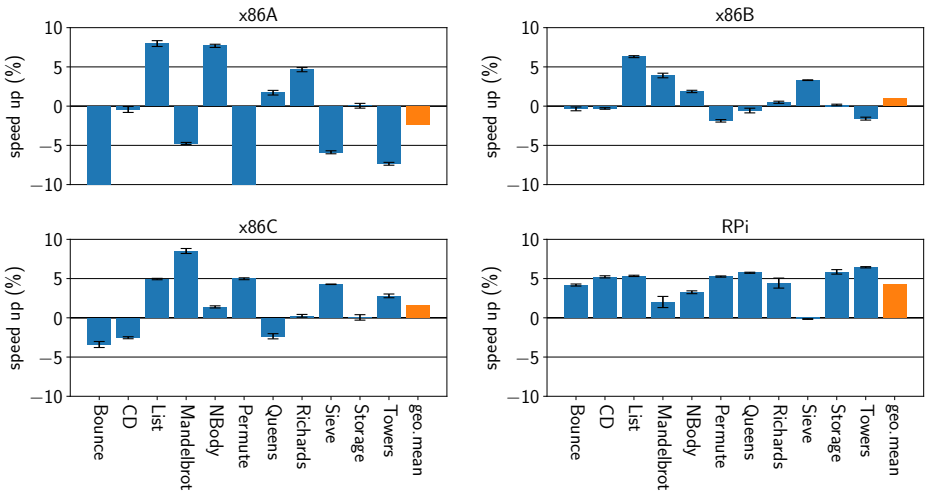

Figure 3: Speed up for the frequency order VMs over the bytecode-number order VMs for four processors (see Section 4 for the details of processors), with 95% confidence interval; the higher, the better. For three out of four machines, speed up was observed on average. However, the divergence implies that other factors beside code locality affect performance.

$f(X)$ with $f(f(X))$. It continues to apply f until the result becomes different from any element in the swapped part.

More precisely, for the parent chromosomes C_1 and C_2 and the random indices i and j where $i < j$, it defines a partial function f such that

$$f = \{C_2[k] \to C_1[k] | i \le k \le j\}.$$

Then, it produces a new chromosome C' for the next generation such that

$$C'[k] = \begin{cases} f^*(C_1[k]) & (k < i \text{ or } j < k) \\ C_2[k] & (i \le k \le j) \end{cases},$$

where $f^*(X)$ is the result of as many applications of f to X as necessary for it to be different from any of $S_2[k]$ with $(i \le k \le j)$.

For the adjacency representation, we used the alternating-edge crossover operator [13]. In the chromosome C, a pair of an index i and the value at that index $C[i]$ represents an edge in the adjacency relationship. Thus, the handler for i needs to be directly followed by the handler for $C[i]$. This operator constructs a path covering all nodes by alternatingly taking edges from both parents. For two parent chromosomes, C_1 and C_2, it first randomly chooses a bytecode number, i_0, and decides the next bytecode i_1 such that $i_1 = C_1[i_0]$ by using the edge of C_1. Thus, for the new chromosome C', $C'[i_0] = C_1[i_0]$. Next, it decides the next bytecode i_2 such that $i_2 = C_2[i_1]$ by using the edge of C_2. Hence, $C'[i_0] = C_1[i_0]$. But, if, in the j-th step, i_{j+1} is already in the new chromosome, it chooses i_{j+1} randomly.

3.3 Mutation

We used the exchange mutation operator [13]. We randomly chose an index i, and then chose another index j. The index j was chosen randomly, but we assigned weights so that the indices closer to i are likely to be chosen.

The exchange mutation may not yield a valid chromosome for the adjacency representation. Therefore, for the adjacency representation, we converted it to the path representation before applying the exchange mutation, and converted to the adjacency representation after the mutation.

3.4 Selection

We will experiment with two standard selection operators in Section 4: the *elitism selection* and a combination of the elitism and roulette wheel selections, which we simply call the *roulette wheel selection* in the rest of this paper. The elitism selection selects the best k chromosomes for the population of the next generation.

The roulette wheel selection, we define a numeric *fitness* of a chromosome i, f_i; the higher, the better. The roulette wheel selection selects two best chromosomes. Then, it selects $(k-2)$ chromosomes for the next generation from the remaining chromosomes randomly with the weights of the fitness. More precisely, the possibility for a chromosome i being selected, p_i, is denoted as

$$p_i = \frac{f_i}{\sum_{j \in C} f_j},$$

where C is the set of all chromosomes except for the two best ones.

We defined the fitness of a chromosome i to reflect the execution speed of the VM generated by it. More precisely, we used the complement of the min-max normalized execution time. The measured execution time for the fastest chromosome is $t_{\min}$ and slowest one is $t_{\max}$. With this, the fitness for chromosome i is defined as

$$f_i = 1 - \frac{t_i - t_{\min}}{t_{\max} - t_{\min}},$$

where t_i is the measured execution time for the chromosome i. For the execution time, we averaged the elapsed times for three executions.

3.5 Making VMs from Chromosomes

Given a chromosome, we constructed a VM that has bytecode handlers arranged in source code in the order of the chromosome. More precisely, we reordered the bytecode handlers in the source code of eJSVM written in C. In eJSVM, the bytecode handlers are inlined in the interpreter function as shown in Figure 2; each of them is not a single function but a labeled code block. For complex operations, such as slow path of property accesses, the bytecode handlers call runtime functions.

The handlers of the constructed VMs may not be precisely arranged in the order of the chromosome because the C compiler's optimizer may rearrange basic blocks. If we reordered bytecode handlers in the generated eJSVM binary, or if we generated bytecode handlers in assembly language, we could arrange the bytecode handlers exactly the same order as the solution. However, if we did so, we would have to give up compiler's optimizations. Rather, we chose to generate interpreter's source code in C language, and optimize the solution to the combination of the compiler and processor. We will call such a combination an *environment*.

4 EVALUATION

As a first step of the evaluation, we discuss how we chose the representation and selection operators for the GA. Then, we assess how much the GA solutions improve eJSVM's performance based on ordering bytecode handlers in the various environments. We also evaluated the generality of the GA solutions from two perspectives:

- Cross-benchmark evaluation: does a solution optimized for a particular program improve other programs in the same environment?
- Cross-machine evaluation: does a solution generated in a particular environment improve performance in a different environment?

In our experimentation, we produced five chromosomes for the initial population and population for each generation. From these five chromosomes, we randomly chose a pair and produced two chromosomes by crossover. We repeated this step eight times to produce 16 chromosomes. Then, we gave 5% of the chance of mutation 10 times for each of the 16 chromosomes; each chromosome could be mutated up to 10 times, but the chance of being mutated 10 times was as small as $(0.05)^{10}$. Finally, we added the five chromosomes of the current generation to the 16 chromosomes, and selected five for the next generation from the 21 chromosomes.

4.1 Experimental Environment

For this evaluation, we ran the benchmarks using the processors and the C/C++ compilers listed in Table 1. The CPU clock frequencies were fixed to the frequencies denoted in Table 1 by turning off turbo boost and hyperthreading, and using the performance governor. For the environment x86A, which had a heterogeneous multicore processor, we bound the eJSVM process to a "performance" core. The RPi environment was a Raspberry Pi 3 Model B. The interpreter binary of eJSVM was around 20.7 KB.

We used benchmarks from the Are We Fast Yet [14] benchmark suite. Because our GA measured execution times to evaluate the fitness for solutions, each benchmark had to be executed thousands of times. Thus, we modified the benchmark programs to reduce iterations performed in the program executions. We also selected only benchmark programs that we could modify to complete execution in a reasonable time in eJSVM; we did not use DeltaBlue, Havlak. We did not choose JSON, either, because the original eJSVM could not run it because it used a very long string that was not supported by eJSVM, which is for embedded systems. As a result, the entire experiment completes within a day for each x86 environment and within three days for RPi.

4.2 Choosing the Representation and Selection Operator for the GA

As outlined in Section 3.1 and 3.4, we consider the path and adjacency representations as well as the elitism and roulette wheel selection operators. This gives us four possible combinations for the GA. To choose one, we compare their convergence after 30 generations in the x86A environment.

Figure 4 plots the convergence curves over the 30 generations in x86A. The Y-axis is the normalized elapsed time for the program running on the VM with the best one among the chromosomes in the generation. The numbers are normalized to an initial solution. After 30 generations, we pick the best one from all the chromosomes generated by the four combinations within this 30 generations. The horizontal line indicates the execution time for the best chromosome. The bar chart labeled geo. mean indicates the geometric mean of the execution times for the best chromosome within the 30 generations for each combination.

When comparing path and adjacency representations, the adjacency-representation-based combinations were likely to produce better results regardless of the selection operations. Thus, adjacency representation with either the elitism selection (shortened to AE) or roulette wheel selection (AR) outperformed both path representation variants (PE and PR). Additionally, we conclude from Figure 4 that the quality of the chromosome reaches a relatively stable level after 30 generations.

For the remaining evaluation, we used AE, the combination of adjacency representation and elitism selection, to generate 30 generations. Among the 30 generations, we picked the best chromosome as the representation of the *optimal* solution. Thus, the optimal solution does not necessarily come from the last generation.

4.3 Performance Improvements by GA

To assess the benefit of the optimal solution found by GA, we produced GA solutions for each benchmark in each environment and measured the execution times of the optimization targets, i.e., the benchmark and the environment for which the solution was optimized. Figure 5 shows the speedups for the GA solutions over the original eJSVM, whose bytecode handlers are arranged in the bytecode-number order. Speedups for Sieve and Mandelbrot in x86C were 23.0% and 20.1%, respectively.

We observed that GA solutions improved the execution speed for all benchmarks and for all environments. Although improvements varied from benchmark to benchmark, average improvements over benchmarks were more than 5% for all environments, and that was, in the best case, 12.4% for x86C.

Table 1: Processors and C/C++ compilers used in our experimentation.

name	CPU Model		compiler
x86A	Intel(R) Core(TM) i9-12900	2.4 GHz	gcc 10.3.0
x86B	Intel(R) Core(TM) i7-11700	2.5 GHz	gcc 9.4.0
x86C	Intel(R) Xeon(TM) W-2235	3.8 GHz	gcc 9.4.0
RPi	Broadcom BCM2837 ARMv8 Cortex-A53	1.2 GHz	gcc 10.2.1

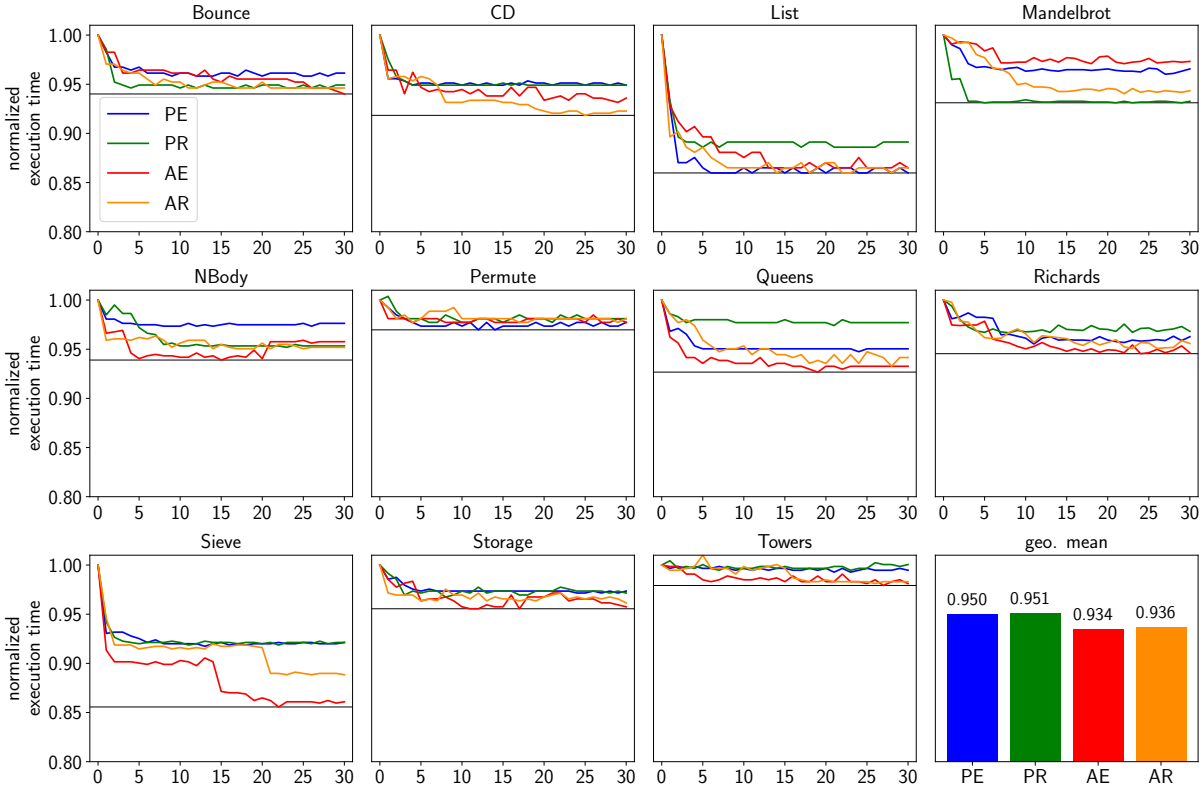

Figure 4: Convergence curve for x86A for four combinations of the solution representations and the selection operators; the path representation (P), the adjacency representation (A), the elitism selection (E), and the roulette wheel selection (R). The X-axis represents the generations.

We conclude that the GA approach was effective to optimize VM to a particular program and a particular environment.

4.4 Cross-Benchmark Evaluation

Next, we assess whether a solution optimized for a specific benchmark can also speed up other benchmarks in the same environment. Outside of embedded scenarios, this is useful since the application is generally not known upfront.

While different programs behave differently at the macro level, they may show similarities at the micro level. Thus, for instance, bytecode sequences may show similar patterns. If this is the case, a solution optimized for a specific program may generalize to other programs and also give speedups. To verify this intuition, we use the solution optimized for a specific benchmark to assess the speedup it gives for the other benchmarks.

The evaluation results in the three x86 environments are shown in Figure 6. Each graph shows the speedups over the original eJSVM by the solution optimized for the program indicated on its title, with 95% confidence intervals; the higher, the better. The X-axes represent the programs that are executed. For example, the second bar in the graph in the upper left corner indicates the speedup for CD for the solution optimized for Bounce. Both the GA search and measurements were conducted in the same environment.

The red bars are results for the solutions optimized for the executed benchmarks. Thus, they should be the same as for the results in Figure 5; because we measured again, the numbers were not exactly the same. The bar geo.mean indicates the geometric mean of all programs, except the one for which the solution was optimized.

A GA solution for one benchmark may not necessarily speed up other benchmarks in the same environment. However, in most

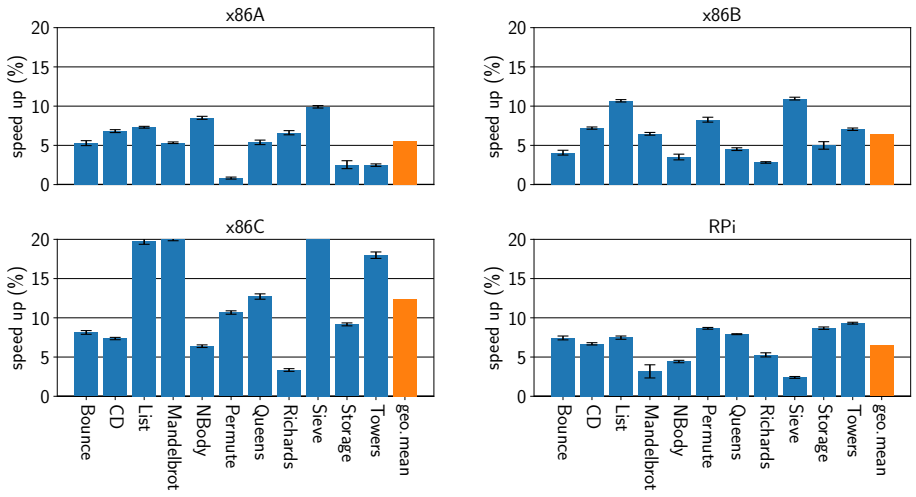

Figure 5: Speed up for the GA solution over the original eJSVM, with 95% confidence interval; the higher, the better.

cases, a GA solution outperformed the original eJSVM regardless of the benchmark for which the solution was optimized. In particular, the experiment in x86C showed speedups for all benchmarks by using any GA solution. On average, all the solutions accelerated different benchmarks from those for which they were optimized, that is, geo.mean indicated positive results, except the solution optimized for List in x86A, which slowed by 0.11%.

Furthermore, some solutions generalize well in terms of speedup across benchmarks. For environment x86A, the solutions optimized for CD, NBody, and Towers sped up most programs, and no program significantly slowed down, i.e., the upper bounds of the 95% confidence intervals were positive. Although these solutions did not speed up Permute, we believe that the original eJSVM was almost optimal for Permute and there was little room to improve in x86A, because Permute did not speed up even by the solution optimized for it, and Permute slowed down by the frequency order as shown in Figure 3. For x86B, Bounce, NBody, Richards, Queens, and Permute were generalized solutions. For x86C, all the programs were sped up with any GA solutions.

We conclude that a GA solution optimized for a particular program is likely to improve execution speed of different programs. Therefore, we could improve VM performance generally by using a GA solution. In this experimentation, we used a single program for the optimization target. We expect we could improve the generality of the solution by using multiple programs for the optimization target. This is one of our future works.

4.5 Cross-Machines Evaluation

We assessed whether a solution optimized for a specific environment can also speed up programs in different environments. If it can, it is feasible to build a single VM that has the order of optimal bytecode handlers in all environments. However, as we show below, the GA solutions did not generalize well across environments. Therefore, we need to optimize the order of the bytecode handlers for each environment to obtain optimal performance.

In this experimentation, we measured the execution times for benchmark programs in all environments using optimized GA solutions for all environments. Figure 7 shows the speedups with 95% confidence intervals; the higher, the better. The bars in each group indicate the speedups for the GA solutions optimized for environments x86A, x86B, x86C, and RPi, from left to right.

For x86A, the GA solutions optimized for different environments delivered slowdowns. For RPi, in contrast, the GA solutions for different environments delivered speedups. For x86B and x86C, the results depended on the solutions. The solution for RPi tended to deliver slowdowns. Because the results varied widely depending on the combination of the environment for which the solution is optimized and the environment in which the program is executed, we conclude that the GA solutions did not generalize well across environments.

5 RELATED WORK

Much previous work uses metaheuristic searches, including GA, to select parameters for compiler optimizations. For example, they were often used to select which optimizations to apply and to determine the order of optimization passes as shown in the survey [1]. Cooper et al. [5] used a GA to decide the sequence of optimization passes that reduced the code sizes of a binary. Kulkarni and Cavazos [12] used neuroevolution of augmenting topologies (NEAT) [20] to determine both the optimizations to apply and their order. Knijnenburg et al. [11] applied an iterative compilation approach to decide the parameters for loop tiling and unrolling in the context of matrix multiplication. They used the iterative compilation approach to find good parameters in an architecturally adaptive manner. These approaches use metaheuristics to decide parameters other than the code arrangement for compiler optimizations.

Optimizations to change the arrangement of code blocks, such as basic blocks and functions, are, however, well-known, too. Pettis and Hansen [19] proposed a profile guided optimization (PGO) of code positioning that optimizes the arrangement of both basic

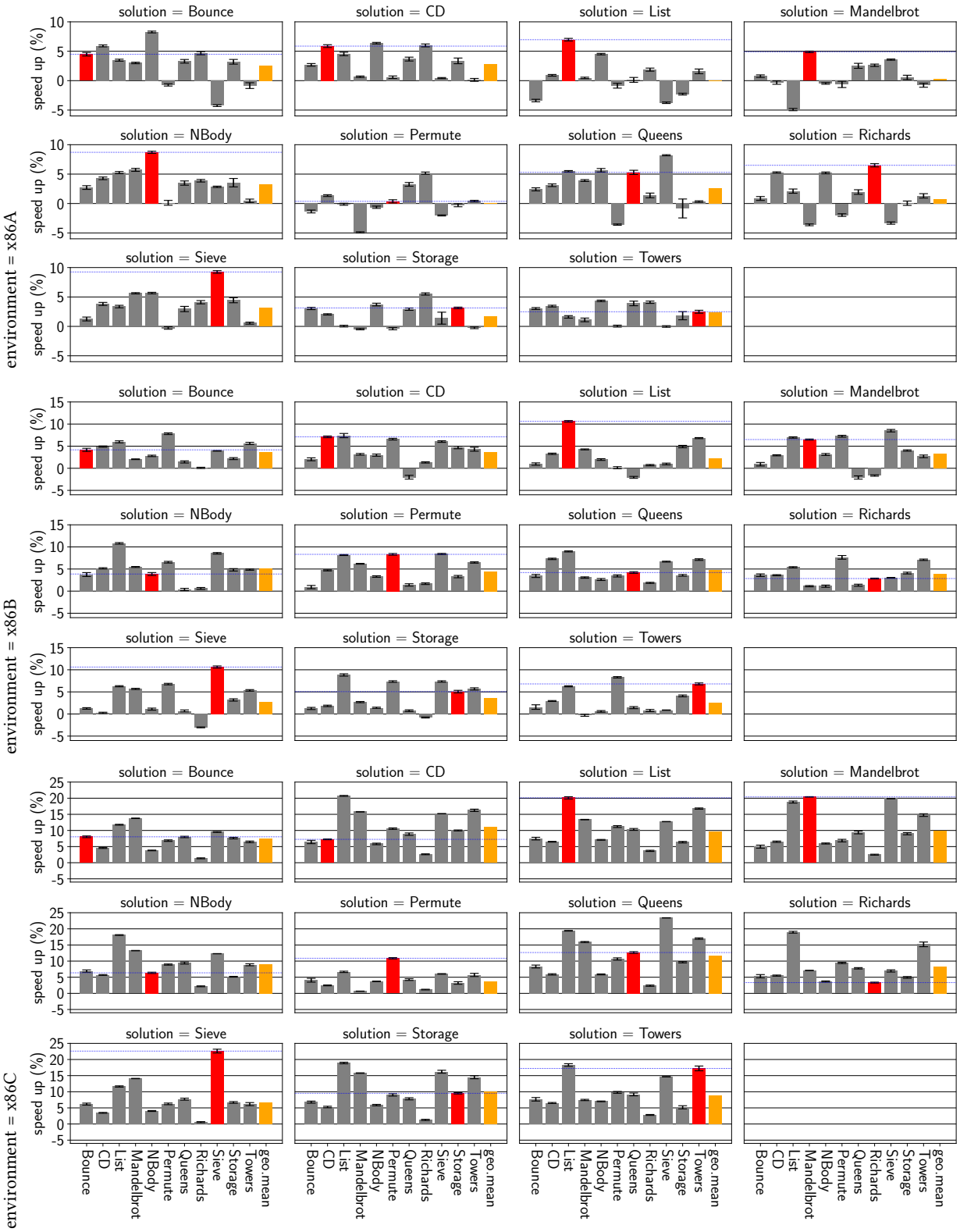

Figure 6: Cross-benchmark evaluation results for three x86 environments.

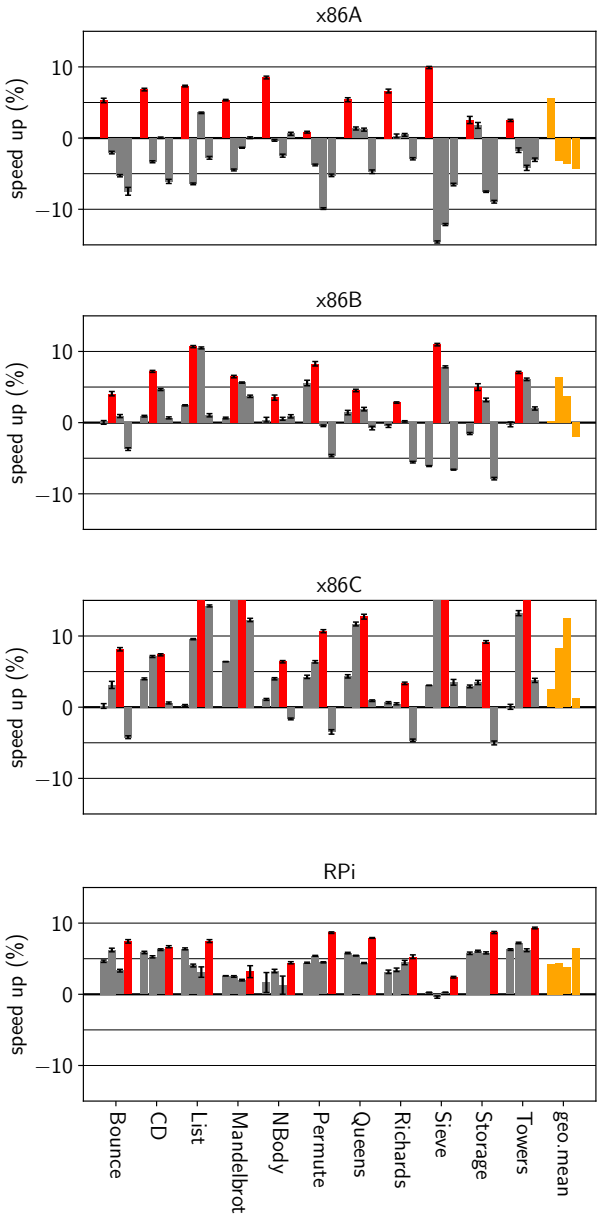

Figure 7: Cross-machines evaluation results. The bars in each group indicate the speedups for the GA solutions optimized for environments x86A, x86B, x86C, and RPi (left to right). A red bar indicates that the program was executed in the same environment as the one the solution was optimized for.

blocks and functions. Their primary goal was to reduce overhead of the instruction memory hierarchy, such as instruction cache misses, and TLB misses. Young et al. [22] proposed a PGO of code arrangement to improve the accuracy of branch prediction. Recent work on PGOs includes Panchenko et al. [18] and Chen et al. [4]. PGO of code arrangement was also well studied in the context of dynamic compilation. Ottoni et al. applied PGO to the JIT compiler of a PHP VM, HHVM [16, 17]. Recently, Newell and Pupyrev [15] used a machine learning approach to reorder basic blocks. These are developed as compiler optimization techniques. In contrast, we applied the reordering of code blocks to a specific software, an interpreter, and modified the source code of the software.

6 CONCLUSION

This paper demonstrated that reordering bytecode handlers of threaded-code interpreters can improve performance, and the GA approach is effective in finding a good handler order. Our GA finds a handler order optimized for a specific program and a specific environment, which is a combination of hardware and a C/C++ compiler. In our experiment using a JavaScript VM, eJSVM, the orders of handlers found by GC substantially improved performance of the program for which the order was optimized when the program was executed in the same environment as the one for which the order is optimized. Furthermore, we found that the orders identified by GA often generalized well in terms of speedup across programs, i.e., the order optimized for a program was likely to speed up other programs, as well. In contrast, they did not generalize across environments. Therefore, it is feasible to develop an optimal interpreter for each processor by searching for an optimal order for it by GA. However, developing a single optimal interpreter for all processors is not feasible with our approach.

Our future work includes to reorder the interpreter binary rather than reordering the interpreter source code so that bytecode handlers reordered by the GA cannot further be reordered by the compiler. Another direction of our future work is to find a more general order across programs by optimizing for multiple programs.

ACKNOWLEDGMENTS

This work was supported by the JSPS through JSPS KAKENHI Grant Numbers JP18KK0315 and JP22H03566, as well as the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

REFERENCES

[1] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2019. A Survey on Compiler Autotuning using Machine Learning. *ACM Comput. Surv.* 51, 5 (2019), 96:1–96:42. https://doi.org/10.1145/3197978

[2] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (1973), 370–372. https://doi.org/10.1145/362248.362270

[3] Kevin Casey, M. Anton Ertl, and David Gregg. 2007. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Trans. Program. Lang. Syst.* 29, 6 (2007), 37–es. https://doi.org/10.1145/1286821.1286828

[4] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proc. International Symposium on Code Generation and Optimization (CGO '16)*. ACM, 12–23. https://doi.org/10.1145/2854038.2854044

[5] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space using Genetic Algorithms. In *Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*. ACM, 1–9. https://doi.org/10.1145/314403.314414

[6] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: statistically sound performance evaluation. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, 219–228. https://doi.org/10.1145/2451116.2451141

[7] Kai Grunert. 2020. Overview of JavaScript Engines for Resource-Constrained Microcontrollers. In *Proc. International Conference on Smart and Sustainable Technologies (SpliTech '20)*. 1–7. https://doi.org/10.23919/SpliTech49282.2020.9243749

[8] Yuta Hirasawa, Hideya Iwasaki, Tomoharu Ugawa, and Hiro Onozawa. 2022. Generating Virtual Machine Code of JavaScript Engine for Embedded Systems. *Journal of Information Processing* 30 (2022), 679–693. https://doi.org/10.2197/ipsjjip.30.679

[9] John H Holland. 1992. Genetic algorithms. *Scientific american* 267, 1 (1992), 66–73.

[10] Takafumi Kataoka, Tomoharu Ugawa, and Hideya Iwasaki. 2018. A Framework for Constructing Javascript Virtual Machines with Customized Datatype Representations. In *Proc. Symposium on Applied Computing (SAC '18)*. ACM, 1238–1247. https://doi.org/10.1145/3167132.3167266

[11] Peter M. W. Knijnenburg, Toru Kisuki, and Michael F. P. O'Boyle. 2003. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. *J. Supercomput.* 24, 1 (2003), 43–67. https://doi.org/10.1023/A:1020989410030

[12] Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '12)*. ACM, 147–162. https://doi.org/10.1145/2384616.2384628

[13] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. 1999. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review* 13, 2 (1999), 129–170. https://doi.org/10.1023/A:1006529012972

[14] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking: Are We Fast Yet?. In *Proc. Symposium on Dynamic Languages (DLS '16)*. ACM, 120–131. https://kar.kent.ac.uk/63815/

[15] A. Newell and S. Pupyrev. 2020. Improved Basic Block Reordering. *IEEE Trans. Comput.* 69, 12 (2020), 1784–1794. https://doi.org/10.1109/TC.2020.2982888

[16] Guilherme Ottoni. 2018. HHVM JIT: a profile-guided, region-based compiler for PHP and Hack. In *Proc. Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 151–165. https://doi.org/10.1145/3192366.3192374

[17] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *Proc. International Symposium on Code Generation and Optimization (CGO '21)*. IEEE, 340–350. https://doi.org/10.1109/CGO51591.2021.9370314

[18] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proc. International Symposium on Code Generation and Optimization (CGO '19)*. IEEE, 2–14. https://doi.org/10.1109/CGO.2019.8661201

[19] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. *SIGPLAN Not.* 25, 6 (1990), 16–27. https://doi.org/10.1145/93548.93550

[20] Kenneth O. Stanley and Risto Miikkulainen. 2002. Efficient Reinforcement Learning Through Evolving Neural Network Topologies. In *Proc. Genetic and Evolutionary Computation Conference (GECCO '02)*. Morgan Kaufmann, 569–577.

[21] Masahiro Yasugi, Yuki Matsuda, and Tomoharu Ugawa. 2013. A proper performance evaluation system that summarizes code placement effects. In *Proc. Workshop on Program Analysis for Software Tools and Engineering (PASTE '13)*. ACM, 41–48. https://doi.org/10.1145/2462029.2462035

[22] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. 1997. Near-optimal Intraprocedural Branch Alignment. In *Proc. Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, 183–193. https://doi.org/10.1145/258915.258932