LII Simpósio Brasileiro de Pesquisa Operacional
João Pessoa - PB, 3 a 5 de novembro de 2020

Microbenchmark Studies in OptFrame: a 10-Year Anniversary

Igor M. Coelho[1], Vitor N. Coelho[3], Anderson Zudio[1], Rodolfo Araújo[2], Matheus N. Haddad[4], Pablo Luiz A. Munhoz[4], Breno S. M. Maia[2], Luiz Satoru Ochi[1], Marcone Jamilson F. Souza[5]

[1] Instituto de Computação - Universidade Federal Fluminense
Rua Milton Tavares S/N, São Domingos, Niterói, RJ
[2] Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 254, CEP 20550-900, Rio de Janeiro, RJ
[3] OptBlocks, Avenida João Pinheiro, 274 Sala 201 - Lourdes, 30130-186, Belo Horizonte - MG
[4] Instituto de Ciências Exatas e Tecnológicas, Universidade Federal de Viçosa - Campus Rio Paranaíba
MG230, km 7, Caixa Postal 22, CEP 38810-000, Rio Paranaíba, MG
[5] Instituto de Ciências Exatas e Biológicas, Universidade Federal de Ouro Preto
CEP 35400-000, Ouro Preto, MG

imcoelho@ic.uff.br, vncoelho@gmail.com, azudio@id.uff.br, rodolfo@ime.uerj.br, matheus.haddad@ufv.br, pablo.munhoz@ufv.br, brenosmmaia@gmail.com, satoru@ic.uff.br, marcone@iceb.ufop.br

RESUMO

OptFrame é um framework em *C++* que facilita a implementação de meta-heurísticas para otimização. Este ano marca o décimo aniversário da publicação do OptFrame no XLII SBPO. Na versão mais recente *4.0*, a ferramenta conta com novas funcionalidades por meio de recursos de padrões recentes do *C++*, como *corrotinas* e *closures*. O OptFrame tem sido utilizado em diversas aplicações da literatura e da indústria, incluindo problemas de roteamento, micro-grids, previsão em séries temporais, marketing direto, escalonamento e geração automática de músicas. Neste trabalho, apresentamos os avanços recentes do OptFrame e um estudo com microbenchmarks para suas componentes básicas, avaliando a sobrecarga computacional. Estudos computacionais mostram que as estratégias propostas foram capazes de reduzir a sobrecarga de 1700% para 34%, através de técnicas avançadas de *Type Erasure* em *C++* aplicáveis a demais *solvers* da literatura.

PALAVRAS CHAVE. Otimização, Framework, Meta-heurísticas, Microbenchmark

ABSTRACT

OptFrame is a *C++* framework that facilitates the implementation of metaheuristics for optimization. This year marks the 10th anniversary of the publication of OptFrame in SBPO XLII. In the most recent *4.0* version, the tool has new functionalities using the latest capabilities of modern C++ standards such as *coroutines* and *closures*. OptFrame has been used in several works of the literature and industry to solve problems such as routing problems, micro-grid, time-series forecasting, direct marketing, scheduling, and automatic music generation. In this work, we show recent improvements in OptFrame with a study based on microbenchmarks for basic components, especially neighborhood exploration. Empirical studies demonstrate that the devised strategies were capable of reducing the original version overhead from 1700% to 34% by advanced techniques of C++ Type Erasure, that can also be applied to other solvers in literature.

KEYWORDS. Optimization, Framework, Metaheuristics, Microbenchmark

1. Introduction

Metaheuristic usage is a common approach to solve practical-scale instances of many optimization problems that are NP-Hard [Gendreau et al., 2010]. Some metaheuristics have similar components to employ its strategy, like local search, solution construction, solution perturbation and recombination. Consequently, algorithm developers may reuse those components through distinct heuristic approaches to various problems. Generally, computational performance and fast development are essential concerns when solving optimization problems. Therefore, there is a demand for frameworks that implement metaheuristics and its typical components for optimization problems.

Many authors have proposed frameworks for optimization problems. Fink e Voß [2003] introduce *HotFrame*, a framework developed in *C++* that provides adaptable components for various metaheuristics and some common problem-specific complements. Di Gaspero e Schaerf [2003] propose a framework called *EasyLocal++*, as an object-oriented framework for the design and analysis of local search algorithms. Cahon et al. [2004] and Liefooghe et al. [2011] present ParadisEO, which is a framework that offers parallel and distributed metaheuristics. A Java framework called jMetal is presented in Durillo et al. [2006] for multi-objective optimization. Giagkiozis et al. [2013] introduce Liger as a framework that is extensible and easy for non-expert usage in the industry. A comprehensive review of frameworks for optimization problems is found in Lopes Silva et al. [2018].

OptFrame was first published ten years ago in SBPO XLII through a work entitled "OptFrame: a computational framework for combinatorial optimization problems" [Coelho et al., 2010]. OptFrame is an open-source project[1] that implements a wide range of features for solving optimization problems with metaheuristics. The original work highlighted the main features of OptFrame, providing benchmarks for both *C++* and Java implementations, as well as usage examples of this framework. Through the years, new features were introduced to aid in the development of algorithms for solving both single and multi-objective problems [Coelho et al., 2016b]. Through these years, the authors conceived multiple improvements and updates to the framework. Several industrial applications use OptFrame to implement their solutions. In the literature, OptFrame was used to devise high-quality applications for several works involving single and multi-objective problems [Souza et al., 2010; Almeida et al., 2011; Munhoz et al., 2012; Coelho et al., 2012, 2016b,a; Munhoz et al., 2018; Zudio et al., 2018].

More specifically, OptFrame is an optimization framework that provides *C++* interfaces for conventional components of trajectory and population-based metaheuristics. It includes efficient implementations of standard versions of the most well recognized and applied metaheuristics. The user can test and adapt them to specific problems. It is possible to fine-tune each component based on problem-specific characteristics. In addition, the framework user has access to quality tools for testing, profiling, debugging and validating each implemented heuristic. A non-exhaustive list of metaheuristics implemented in OptFrame is: Genetic Algorithm [Whitley, 1994], Variable Neighborhood Search (VNS) [Hansen et al., 2017], Tabu Search [Glover e Laguna, 2013] and Greedy Randomised Adaptative Search (GRASP) [Resende e Ribeiro, 2016]. Nowadays, the latest improvements in OptFrame use modern *C++* features like Type Erasure, coroutines and closures.

In the present paper, we present empirical results focused on state-of-the-art microbenchmark technologies, comparing framework-induced overheads from a low-level perspective. The computational experiments show that the original version of OptFrame has approximately 1700% overhead ratio in some of the fundamental components applied in neighborhood exploration, while the newer version has 34%. This work describes how these results were obtained, while reviewing

[1] OptFrame official repository is available at https://github.com/optframe/optframe.

advanced techniques applied with C++ Type Erasure. These techniques can also be applied to works out of the OptFrame context, while being useful to any application that works with generic programming, like other solvers in the literature for optimization problems. These results also motivated future changes in OptFrame, trying to achieve high performance while providing useful programming abstractions.

The rest of this work is structured as follows: Section 2 details some of the concepts used through this paper, Section 3 gives an overview of the core components of OptFrame, Section 4 shows the empirical experiments performed with OptFrame using microbenchmarks and Section 5 concludes the work alongside future perspectives.

2. Key Concepts and Applications

Before introducing some of the essential components of the framework, this section briefly describe main concepts that are applied in the context of OptFrame.

2.1. Optimization Problem, Solution Space and Objective Space

An optimization problem Π is a quadruple $(I, \mathcal{Z}_\prec, \mathcal{S}, z)$, where I is the set of instances, $\mathcal{Z}_\prec$ is the objective space with order relation $\prec$, an instance $\pi \in I$, $\mathcal{S}(\pi)$ is the set of solutions for π, where $\mathcal{S}_f \subset \mathcal{S}(\pi)$ is the set of *feasible solutions*[2], and *objective function* $z : \mathcal{S}_f \mapsto \mathcal{Z}_\prec$.

2.1.1. Single Objective Problem

In the literature, most classical problems are *single objective problems* (SOP) that considers a single optimization direction. For this definition, we consider (without loss of generality) an objective space $\mathcal{Z}_\leq \subset \mathbb{R}$ with *total order*[3] relation $\leq$ (for *minimization* problems).

Given a solution s in space $\mathcal{S}_f$ with *objective value* $z(s) \in \mathbb{R}$, the goal is to find an *optimal solution* $s^* \in \mathcal{S}_f$ with $z(s^*) \in \mathbb{R}$, such that $z(s^*) \leq z(s'),\ \forall s' \in \mathcal{S}_f$:

$$\begin{aligned} (SOP)\ \ minimize\ \ & z(s) \\ s.t.\ \ & s \in \mathcal{S}_f \end{aligned} \tag{1}$$

2.1.2. Multi-objective Problem

We also consider multi-objective problems (MOP), where multiple objective functions z_k, $k = 1, \cdots, p$, are taken into account to model the problem. Since these objectives may have conflicts, the notion of *optimal solution* does not hold anymore for a MOP.

In MOP literature, the objective space $\mathcal{Z}_\prec \subset \mathbb{R}^p$ is typically defined with a *partial order*[4] relation $\prec$ known as *Pareto dominance*, where $x \prec x'$ (x *dominates* x') if, and only if, $z_k(x) \leq z_k(x')\ \forall k \in \{1, ..., p\}$, and for some k the inequality is strict. We say that a solution $s^* \in \mathcal{S}_f$ is *efficient*, if there is no $s' \in \mathcal{S}_f$ such that $z(s') \prec z(s^*)$. A MOP definition from Lust e Teghem [2010]:

$$\begin{aligned} (MOP)\ \ minimize\ \ & z_k(s)\ \ k = 1, ..., p \\ s.t.\ \ & s \in \mathcal{S}_f \end{aligned} \tag{2}$$

We denote $z(s^*) \in \mathbb{R}^p$ as a *non-dominated point*, and the collection of all efficient solutions is called *efficient set* $\mathcal{P}^* \subset \mathcal{S}_f$, where its image in the objective space is a *Pareto front*.

[2] Note that some optimization techniques consider the exploration of a solution space $\mathcal{S}$ comprising both feasible and some unfeasible solutions. On practice, most of these methods transform the original problem into a relaxed version that allows some unfeasible solution to become feasible, by *paying some extra costs*. These are not covered in this work.

[3] A *total order* must fulfill the following properties: antisymmetric, transitive and connex relation.

[4] We assume a *partial order*, replacing the *connex relation* from a *total order* by an *irreflexivity* property (also called *strict partial order*). In some works in literature, the term *Weak Pareto Dominance* is defined as a *non-strict partial order* with *reflexivity* property, which is not the case here.

2.1.3. Other Order-Based Evaluations

Many works in literature deal with several objectives, however with a *pre-defined priority* between them (which is not the case for general MOP). By respecting original ordering on p objectives for two solutions s and s', one can assume a *total preorder*[5] relation $\lesssim$. An example of such strategy can be seen on Sousa et al. [2015], for a TSP with Hotel Selection:

$$z(s) \lesssim z(s') \text{ iff } \{\exists k \in \{1, ..., p\} : \forall l < k, \ z_l(s) = z_l(s') \text{ and } z_k(s) < z_k(s')\} \quad (3)$$

A variant of $\lesssim$ can be found on Dubois et al. [2001], where a lexicographic ordering $>^{lex}$ is proposed for fuzzy optimization problems. One can also use scalarization techniques [Talbi, 2009] with weights $w_k \in \mathbb{R}$ to generate a new single objective from multiple objectives, such that, $z(s) = \sum_k^p w_k \cdot z_k(s)$ (useful when no conflicts exist between objectives). Finally, decision problems can be seen as an special case of SOP, where $z(s) \in [0, 1]$ represents answers *no* and *yes*.

2.1.4. Solutions and Evaluations on OptFrame

The concepts described before are easily implemented in OptFrame, using standard framework classes. A `Solution<R>` implements a solution space element for any type `R` (being it a permutation, binary vector, graph, etc). An objective space element `Evaluation<Z>`, where type `Z` describes the underlying type for objective space. The class `Evaluator<Z>` implements the operator $\prec$ as comparison methods `betterThan` and `equals`, for any given type `Z` (it is set by default to C++ type `double` for any SOP).

2.2. Neighborhood Structures

The concept of *neighborhood structure* is central to OptFrame metaheuristics, specially to those with any concept of *local search* [Talbi, 2009]. As in Hansen et al. [2017], the set $\mathcal{N}(s)$ represents *move* operators $\{m_1, m_2, ...\}$ capable of transforming a given solution s into *neighbor* solutions s', i.e., for a given $m \in \mathcal{N}(s)$, $s' = m(s)$. The move operation can be applied to a solution, and eventually be *undone* by a *reverse move* $\bar{m}$, such that, $s = \bar{m}(s')$, for a given neighbor s'. For a SOP with evaluation function z, a move m has an associated *cost* calculated as $\widehat{m}(s) = z(s) - z(m(s))$.

The calculation of a move cost can be provided by user (in an efficient manner), or automatically performed by OptFrame using a successive *apply* and *undo* operation for a given move. If no *undo* operation is provided by the user for a given move, OptFrame is able to copy the solution s, generate its neighbor $s' = m(s)$, and then compute the cost $\widehat{m}(s) = z(s) - z(s')$, without destroying the original solution s (in a less efficient implementation). As discussed in recent works [Nascimento Silva et al., 2020], we also assume that moves can be composed $m_3 = m_1 \circ m_2$ if they are *independent*, i.e., their costs are kept the same when applied in any order: $\widehat{m_3}(s) = \widehat{m_1}(s) + \widehat{m_2}(s), \ \forall s \in \mathcal{S}_f$.

2.2.1. Neighborhood Exploration Primitives

To explore a given neighborhood, some primitives are typically considered for improvement (local search) heuristics. We assume $\mathcal{N}(s)$ behaves like a *stream* of moves for solution s, and we may wish to: (a) find *any* improving move, $findAny(s) = \{m \mid m \in \mathcal{N}(s) \wedge \widehat{m}(s) \leq 0\}$; (b) find the *first improving move*, $findFirst(s) = \underset{m_k \in \mathcal{N}(s) \wedge \widehat{m_k}(s) \leq 0}{\arg\min} k$; (c) find the *best improving move*, $findBest(s) = \underset{m \in \mathcal{N}(s) \wedge \widehat{m}(s) \leq 0}{\arg\min} \widehat{m}(s)$. The *findAny* is very similar to *findFirst*, while the former is assumed to be *non-deterministic* and the latter is expected to be *deterministic*.

[5] We assume a *total preorder* with the following properties: transitivity, connexity and reflexity.

2.2.2. Neighborhood Structure Applications on OptFrame

There are three types of neighborhoods in OptFrame: `NS` class abstraction generates a random `Move` instance m, i.e., $m \in \mathcal{N}(s)$ (thus providing primitive *findAny*); `NSSeq` is a list iterator that generates every possible `Move`, i.e., $m_k \in \mathcal{N}(s)$, $\forall k \in \{1, 2, ...\}$ (providing both *findFirst* and *findBest*, besides *findAny* primitive); `NSEnum` is a random-access iterator for any h-indexed `Move`, i.e., $m_h = \mathcal{N}(h)$, $h \in \{1, 2, ...\}$ (provides all three fundamental *find* primitives).

Stochastic metaheuristics such as Simulated Annealing [Kirkpatrick et al., 1983] and Late Acceptance Hill-Climbing [Burke e Bykov, 2017] use `NS` to provide random moves and explore the solution space. Metaheuristics that depend on local search [Gendreau et al., 2010], such as ILS, VNS and Tabu Search [Glover e Laguna, 2013], typically require a `NSSeq`, in order to provide systematic *neighborhood exploration*. This exploration can be done by varying strategies, such as classical *Best Improvement*, *First Improvement* and *Random Sampling*. Another recent exploration technique is the *Multi Improvement* [Rios et al., 2016; Nascimento Silva et al., 2020], that provides combination of independent moves during exploration. Stochastic components on other metaheuristics can also use the concept of `NS` to generate random mutations (Genetic, Memetic Algorithms and multi-objective variants like NSGA-II [Deb et al., 2002]), perturbation/shaking (ILS and VNS), etc.

2.3. Constructive Methods and Specific Abstractions

Some abstractions are also provided for construction-based metaheuristics, such as Greedy Randomized Adaptive Search Procedures (GRASP) and its variations [Gendreau et al., 2010]. Other metaheuristics, such as Genetic Algorithms and Evolutionary Algorithms on general may require other specific abstractions (such as *crossover*), allowing users to configure method-specific parameters (we refer the reader to OptFrame website for specific details of all supported methods).

3. Modern Programming Techniques

Frameworks propose abstractions that significantly reduce the burden over users to test and experiment different algorithms. But this generalization always comes with a cost. One way to prevent such undesired costs is to allow users to specify every component when needed: for example, on OptFrame, one can design general neighborhood iterators and use several heuristics, or simply code a local search heuristic "manually". One side effect of such choice is that users tend to be quickly satisfied with their "working version" (which is the purpose of a framework) and assume that compiler optimizer would finish the job. But will it manage to do it? At what extent? Is there any "inherent" overhead brought by framework abstractions?

OptFrame performs these *generalizations*, from a *problem-specific component* into a *general component*, by using standard C++ object-oriented inheritance. On practice, we apply the *Liskov Substitution Principle* [Liskov, 1987] to guarantee that every component sub-type behaves correctly, given its expected interface. For this reason, OptFrame is able to provide complex metaheuristic strategies, by only requiring basic operations from the user, e.g, Simulated Annealing and stochastic searches if user provides `Move` abstractions, and ILS/VNS searches if user provides basic `NSSeq` iterators. In this direction, some handy programming techniques from the 60's are only becoming widespread recently, which is the case of *coroutines*.

3.1. Coroutines and Microbenchmarks

Coroutines are a generalization of subroutines, as they *"may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines"* [Conway, 1963]. A coroutine can resume its own execution and transfer the processor usage to another module, that characteristic makes it very useful when implementing non-preemptive applications, cooperative tasks, event loops, infinite iterators, pipes, and iterators (or *generators*).

Cooperative behavior makes a coroutine a good choice for I/O bound applications, by programming it asynchronously it is possible to obtain multiprogramming without preemption thereby simulating concurrent tasks with no need to lock resources. Applications in the context of operations research are mostly CPU bound, although the recent usage of parallelization techniques naturally creates an async environment. Applications such as the research in Araujo et al. [2018] use GPU programming to process multiple tasks in parallel what creates an async communication between the subprocesses. In the face of modern applications where heterogeneous and edge computations resources are used across the network the CPU bound application increases the I/O operations due to the network interaction [Araujo et al., 2020].

All these techniques need to be properly tested when put on practice, in order to ensure that the expected effects are actually felt by the user of application. Some features of OptFrame involves many calls to key operations that, although quite fast, these tiny costs may accumulate over hundreds of thousands of iterations. Modern microbenchmarking tools allow precise timing of small operations, and by repeating the process thousands of times, some undesired timing effects from operating system processing components can be mitigated.

4. Computational Experiments

An empirical study was conducted with microbenchmarks in order to verify the overhead of OptFrame structures for basic operations. The computational environment used is equipped with an *Intel(R) Core(TM) i5-7200U CPU 2.50GHz*, 8 GB of RAM, and Linux kernel *4.15.0-29deepin-generic*. Applications were compiled with *GCC 10.1* from the GNU Compiler Collection and *Clang 11*, using flags `-Ofast`, `-fno-rtti`, `-fno-exceptions`, and C++20 standard. For microbenchmarks, we used *google-benchmark*[6] framework, with disabled CPU frequency scaling, and using optimizing avoidance `ASM volatile` techniques to prevent code from being optimized away. The study involved the investigation of 63 different modeling strategies[7], applying the programming techniques described in Section 3 with over 5000 lines of C++ code specifically designed for these benchmarks[8], on OptFrame Functional Core 4.1.

4.1. Target problem: a $findBest$ strategy

The first experiments focused on the neighborhood abstraction, comparing a *findBest* primitive written in "pure" C/C++ against the equivalent OptFrame implementation. We chose a quadratic neighborhood inspired by swaps in a classic Traveling Salesman Problem (TSP) [Applegate et al., 2006]. The expected number of move operations in $findBest$ is quadratic over a problem with size N (number of cities in TSP). For each move, we apply it to a random initial solution vector and then revert it to original vector (via undo operation), and ensure that operation is correctly performed and not removed during optimization phase. We considered varying sizes of the target problem N from 10, 20, 30, 100, 200 to 1000 (since the neighborhood is quadratic, the expected number of operations ranged from 100 up to $1,000,000$). The number of runs is defined automatically by *google-benchmark*, and average results are presented in *nanoseconds*.

4.2. C/C++ baseline

The C/C++ baseline code is presented in Listing 1, including a C++20 coroutine `co_yield` counterpart. It has been designed to use only C language features to keep it as efficient as possible (although the *google-benchmark* library itself is written in C++).

[6]Google benchmark repository is available at https://github.com/google/benchmark.
[7]Initially 54 for triage, then 9 final benchmark sets.
[8]The benchmark set is publicly available at https://github.com/optframe/optframe.

Listing 1: C++ baseline for $findBest$ microbenchmark implementation

```
unsigned N = state.range(0);
unsigned seed = state.range(1);
for(auto _ : state) { // many tests
  state.PauseTiming();
  auto esol = rand_sol(N, seed);
  state.ResumeTiming();
  //quadratic number of moves O(N^2)
  for(int i = 0; i < N - 1; ++i) {
    for(int j = i + 1; j < N; ++j) {
      // swap operation (apply)
      int aux = esol[i];
      esol[i] = esol[j];
      esol[j] = aux;
      // 'asm volatile' read 'esol'
      // undo swap
      int aux2 = esol[i];
      esol[i] = esol[j];
      esol[j] = aux2;
    }
  }
}
```

Listing 2: C++20 coroutine `co_yield` implementation of $findBest$ for move generator

```
generator< pair<int,int> > swaps(int N){
  for(int i = 0; i < N - 1; ++i)
    for (int j = i + 1; j < N; ++j)
      co_yield make_pair(i, j);
}
// ... then use neighborhood generator
int N = state.range(0); // from tests
unsigned seed = state.range(1);
auto esol = rand_sol(N, seed);
// quadratic number of moves O(N^2)
auto iter = swaps(N); // get generator
while(iter.next()) {
  auto [i, j] = iter.getValue();
  // perform swap operations with (i,j)
  int aux = esol[i];
  esol[i] = esol[j];
  esol[j] = aux;
  // 'asm volatile' read 'esol'
  // undo swap
  int aux2 = esol[i];
  esol[i] = esol[j];
  esol[j] = aux2;
}
```

This proposed baseline achieved the lowest execution time, as expected, from all 63 considered configurations. Table 1 presents the C/C++ baseline results for compilers GCC and clang.

Table 1: C/C++ baseline on compilers GCC and clang

	Average Time (ns) / # Tests					
N	**10**	**20**	**30**	**100**	**200**	**1000**
GCC-10.1	559 / 1251606	940 / 737024	1606 / 436090	14353 / 48886	55187 / 12661	1348109 / 519
Clang-11	542 / 1303859	913 / 769006	1576 / 454071	14090 / 49808	55962 / 11966	1414863 / 495
Difference	-3.14%	-2.96%	-1.90%	-1.87%	1.40%	4.95%

From Table 1, we observe that clang is 3.14% better than GCC for smaller N, but significantly worse for larger N. For other benchmarks, we also observed experiments around 8% slower on clang, up to 100% on heap-allocation experiments, when compared to GCC. For this reason, we selected GCC output binary as our C/C++ baseline.

4.3. Comparison with OptFrame FCore 4.1

The initial experiment consisted in comparing the C/C++ baseline with an equivalent implementation in OptFrame Functional Core 4.1. Table 2 shows the overhead (over baseline) for standard FCore neighborhood enumeration strategy for $findBest$.

As shown in Listing 2, coroutines are handy at the development of iterator-like abstractions (in a concept named *generators*), so we experience these newest C++20 capabilities on OptFrame. We considered GCC 10.1 with same configuration as before, with flag `-fcoroutines`.

From Table 2, we observe that FCore 4.1 has overheads varying from 4x to 17x. Although C++ proposes a "zero overhead" principle on general, the observed overhead was still significant on OptFrame $findBest$, despite the "simplicity" of its components. Coroutine overheads were modest, with 3.4% on average for larger instances.

Table 2: Overhead over baseline for FCore default neighborhood enumeration

	Average Time (ns) / Overhead over baseline (%) / Coroutine overhead (%)			
N	**10**	**20**	**100**	**200**
baseline	559 / 0 / 0	940 / 0 / 0	14353 / 0 / 0	55187 / 0 / 0
FCore 4.1	2825 / 405.4 / 0	10262 / 991.7 / 0	250624 / 1646.1 / 0	1001241 / 1714.3 / 0
coroutine	3026 / 441.3 / 7.1	10786 / 1047.4 / 5.1	259298 / 1706.6 / 3.5	1035705 / 1776.7 / 3.4

It is worth mentioning that such overheads may not be so evident when the $findBest$ cost calculation is more expensive, a common case for practical problems, as this would likely mitigate most of the iterative overhead. On the other hand, it is challenging to develop "pure" C/C++ baselines for comparison on complex problems, as the purpose of a framework is the simplification of the implementations, even if some overhead is incurred. In order to discover the origin of these overheads, and to propose solutions for them, we conducted deeper analysis on the generated code.

4.4. Reducing the overheads: from 1700% to "*zero*"

The iterative structure on FCore 4.1 consists of an iterator that generates heap-allocated `Move` instances (described on Section 2.2). To handle these instances, OptFrame uses ownership managed memory by C++11 standard `std::unique_ptr`, considered to be the closest approach to a zero-overhead abstraction over general pointers [Meredith, 2009].

In order to cancel the 1700% overhead, we needed to change some abstractions. The first change was to completely avoid heap allocations, only using stacked objects. Dozens of proposals were developed and tested, until we finally found solutions that managed to reduce the overhead. Table 3 presents the specific strategies (with partially hardcoded loops) explored in this work.

Table 3: Specific strategies and their respective times (in ns) and overheads (in $\%$) over baseline

	Average Time (ns) / Overhead over baseline (%)				
N	**10**	**20**	**30**	**100**	**200**
baseline	559 / 0.00	940 / 0.00	1606 / 0.00	14353 / 0.00	55187 / 0.00
unique move	1343 / 140.25	4151 / 341.60	8550 / 432.38	89886 / 526.25	374217 / 578.09
multi capture	1131 / 102.33	3124 / 232.34	6575 / 309.40	67846 / 372.70	266345 / 382.62
single capture	723 / 29.34	1704 / 81.28	3161 / 96.82	31978 / 122.80	119703 / 116.90
single static	606 / 8.41	1115 / 18.62	1923 / 19.74	17627 / 22.81	70163 / 27.14

Strategy *unique move* reduces the global overhead by four times (from 4x-17x to 1.4x-5.7x), by allocating an unique global state for a stack-based `Move` object abstraction (polynomial reduction on memory). The downside of such approach is that the instances behave like a *singleton*, limiting more complex move composition strategies, such as the Multi Improvement [Nascimento Silva et al., 2020]. After that, we observed that overheads were reduced to 1x-3.8x when class functions are hardcoded as *capture lambdas* (on *multi capture* approach), instead of an inherited instance. The type erasure costs from each of these capture lambdas seemed to accumulate quickly, so we moved to a strategy where only a *single capture* function was "shared" as reference among all `Move` objects, reducing overheads to 0.29x-1.2x. At this point, we have realized that any tiny overhead on the `Move` abstraction could have significant implications during neighborhood exploration (since the operation is repeated thousands of times).

One interesting approach is presented on the last line of Table 3, where we observe that a single non-stack access (even by global/static methods) already incur in some minimal overhead (from 8% to 27%). So, by allocating even a tiny amount of space in global scope, it generates some

overhead to read and update them (during iterative process). Although we did not verify assembly code (using *perf* tools), it could be that this tiny difference (in nanosecs) represents some extra costs related to typical *data segment* access/allocation (for global variables), when compared to local variables (typically put on local stack). So, *single static* approach acted as a lower bound into our overhead-reducing capabilities.

Table 4 describes the final (fine-tuning) improvements on `Move` function allocation on generic strategies (by using C++ `std::function` techniques without any hardcoded parts) and their respective overheads (these generic strategies were directly tested on OptFrame[9]).

Table 4: Fine tuning function storage types for generic move operations: performance analysis

Average Time (ns) / Overhead over baseline (%)						
N	**10**	**20**	**100**	**200**	**1000**	$\times O(N^2)$
baseline	559 / 0	940 / 0	14353 / 0	55187 / 0	1348109 / 0	1.3
Func. Cpy.	1048 / 87.5	3142 / 234.3	71286 / 396.7	287172 / 420.4	7309379 / 442.2	6.1
Func. Ref.	658 / 17.7	1649 / 75.4	30446 / 112.1	129101 / 133.9	3065785 / 127.4	2.4
Func. Ptr.	606 / 8.4	1113 / 18.4	18014 / 27.5	70248 / 27.3	1813828 / 34.5	1.7

From Table 4, strategy *Function Copy* provides a stateful strategy where each move operation has its own state, and this fact implies much higher overheads over baseline. We observe that strategy *Function Reference* has overheads reduced to 0.17x-1.2x, by completely removing the local state abstraction (with a move data structure uniquely allocated in memory), and only providing references to methods allocated statically (Move "class" has no methods or fields at this point, only references to stateless lambdas). However, even these references seemed to imply some costs, so strategy *Function Pointer* finally is able to achieve expected lower bound overheads of 8%-34%, by providing only copies of pointers to method functions (for some unknown reason, these operated faster than C++ references). By default, *google-benchmark* tries to estimate the complexity of the benchmark in Big-O notation (by the `Complexity()` benchmark parameter) for the following classic growth functions: $O(1)$, $O(N)$, $O(N^2)$, $O(N^3)$, $O(logN)$ and $O(NlogN)$. As expected, *google-benchmark* automatically detected the baseline code as being a $O(N^2)$ function, and gave estimation parameters for it: 1.3 $O(N^2)$.

Finally, Table 5 describes the state characteristics of each function allocation strategy: from stateful *Function Copy* to stateless *Function Pointer*.

Table 5: Fine tuning function storage types on C/C++ and feature details

Strategy	**Stateful**	**Closure**	**C/C++ Function Storage Type**
Function Copy	yes	yes	`std::function<bool(solution_type&)>`
Function Reference	no (*shared*)	yes	`std::function<bool(solution_type&)>&`
Function Pointer	no (*unknown*)	no	`bool(*func)(solution_type&)`

It is worth mentioning that all these strategies can be transparently provided to the user, since FCore 4.1 already deals with a functional programming abstraction in C++ (user only provides the functions, no matter how these are "glued" into OptFrame classes). Also, users are also capable of overwriting any implementation (for project-specific abstractions since early versions of OptFrame), so these guidelines serve as a path for users achieving this 1700% reduction in overheads, even when default implementations does not explore them (being also applicable to any other object-oriented C++ optimization solver in literature).

[9]On practice, these were tested on an alternative "experimental fork" called AltFrame (inside OptFrame repo tests).

5. Conclusions and Future Works

In this work, we celebrate the 10-year anniversary of OptFrame first publication in SBPO XLII. As a framework aiming meta-heuristics and optimization techniques, we cover its recent improvements with latest C++ features (such as coroutines). We evaluate the overhead caused by framework abstractions using a microbenchmark perspective, and compared to a pure C/C++ baseline implementation of a $findBest$ neighborhood exploration primitive. During our research experimentation, we discovered that there may be some inherent overhead, even if it is very small, in abstracting neighborhood iterative processes. A standard object-oriented implementation have presented a 1700% overhead over baseline, while C++ coroutines increased extra 3.4%. Using advanced type erasure techniques, it was possible to reduce these overheads to 8%-34%.

It is worth mentioning that, from our experiments, this 8%-34% overhead is naturally introduced in any C/C++ reference code, as long a global/static variable is used (which is a common practice for C/C++ programmers). So, we strongly believe that for any realistic implementation, even with the simplest programming features, it is very likely to suffer the same overhead (programmers typically will not be able to put all variables in stack/registers). For this reason, we consider this 8%-34% overhead acceptable as part of OptFrame neighborhood iteration structures (although it could be theoretically eliminated if user directly coded and integrated the intended algorithm without any given abstraction).

For the future, OptFrame should continue to efficiently incorporate novel techniques and abstractions, while taking advantage of the concepts described in this paper (from a microbenchmark perspective). More documentation and examples is also welcome, in order to attract new users and to soften the learning curve. Specially, from a hybrid-metaheuristics and hyper-heuristics perspective, the OptFrame looks promising, as the user only needs to fine-tune parameters from a wide range of available components, without needing to coding and testing them. All the techniques described can also be used in any other framework in literature, including the microbenchmarking and coroutine generator. So, from our perspective, this is a very fortunate contribution of this work.

Acknowledgements

OptFrame development was partially supported over these years by Brazilian funding agencies CAPES (finance code 001), CNPq, FAPERJ and FAPEMIG. A special thanks for the support from Brazilian universities UFOP, UFF, UERJ and UFV involved in this project.

Referências

Almeida, B. M. A., Coelho, V. N., Toffolo, T. A. M., Coelho, I. M., Souza, M. J. F., e Coelho, B. N. (2011). MSGVNS: Um algoritmo heurístico para o problema de gerenciamento de escala operacional de controladores de tráfego aéreo. In *Anais do X SITRAER*, p. 154–163, Ouro Preto.

Applegate, D. L., Bixby, R. E., Chvatal, V., e Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton university press.

Araujo, R. P., Coelho, I. M., e Marzulo, L. A. J. (2020). A multi-improvement local search using dataflow and gpu to solve the minimum latency problem. *Parallel Computing*, p. 102661. https://doi.org/10.1016/j.parco.2020.102661.

Araujo, R. P., Rios, E., Coelho, I. M., Marzulo, L. A., e Castro, M. C. (2018). A novel list-constrained randomized vnd approach in gpu for the traveling thief problem. *Electronic Notes in Discrete Mathematics*, 66:183–190.

Burke, E. K. e Bykov, Y. (2017). The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78.

Cahon, S., Melab, N., e Talbi, E.-G. (2004). ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380.

Coelho, I. M., Ribas, S., Perche, M. H. P., Munhoz, P. L. A., Souza, M. F., e Ochi, L. S. (2010). Optframe: a computational framework for combinatorial optimization problems. In *XLII Simpósio Brasileiro de Pesquisa Operacional*, p. 1887 – 1898, Bento Gonçalves, RS.

Coelho, V. N., Souza, M. J. F., Coelho, I. M., Guimaraes, F. G., Lust, T., e Cruz, R. C. (2012). Multi-objective approaches for the open-pit mining operational planning problem. *Electronic Notes in Discrete Mathematics*, 39:233 – 240.

Coelho, V. N., Coelho, I. M., Coelho, B. N., Reis, A. J., Enayatifar, R., Souza, M. J., e Guimarães, F. G. (2016a). A self-adaptive evolutionary fuzzy model for load forecasting problems on smart grid environment. *Applied Energy*, 169:567 – 584.

Coelho, V., Grasas, A., Ramalhinho, H., Coelho, I., Souza, M., e Cruz, R. (2016b). An ils-based algorithm to solve a large-scale real heterogeneous fleet {VRP} with multi-trips and docking constraints. *European Journal of Operational Research*, 250(2):367 – 376.

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408.

Deb, K., Pratap, A., Agarwal, S., e Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.

Di Gaspero, L. e Schaerf, A. (2003). Easylocal++: an object-oriented framework for the flexible design of local-search algorithms. *Softw. Pract. Exper.*, 33(8):733–765.

Dubois, D., Fortemps, P., Pirlot, M., e Prade, H. (2001). Leximin optimality and fuzzy set-theoretic operations. *European Journal of Operational Research*, 130(1):20–28.

Durillo, J. J., Nebro, A. J., Luna, F., Dorronsoro, B., e Alba, E. (2006). jMetal: A java framework for developing multi-objective optimization metaheuristics. Technical Report ITI-2006-10, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, Campus de Teatinos.

Fink, A. e Voß, S. (2003). Hotframe: A heuristic optimization framework. In Vob, S. e Woodruff, D., editors, *Optimization Software Class Libraries*, volume 18 of *Operations Research/Computer Science Interfaces Series*, p. 81–154. Springer US.

Gendreau, M., Potvin, J.-Y., et al. (2010). *Handbook of metaheuristics*, volume 2. Springer.

Giagkiozis, I., Lygoe, R. J., e Fleming, P. J. (2013). Liger: an open source integrated optimization environment. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation (GECCO'13)*, p. 1089–1096, Amsterdam. ACM.

Glover, F. e Laguna, M. (2013). Tabu search. In PARDALOS, M. P., DU, D.-Z., e GRAHAM, R., editors, *Handbook of Combinatorial Optimization*, p. 3261–3362. Springer, New York, 1 edition.

Hansen, P., Mladenović, N., Todosijević, R., e Hanafi, S. (2017). Variable neighborhood search: basics and variants. *EURO Journal on Computational Optimization*, 5(3):423–454.

Kirkpatrick, S., Gelatt, C. D., e Vecchi, M. P. (1983). Optimization by simulated annealing. *science*, 220(4598):671–680.

Liefooghe, A., Jourdan, L., e Talbi, E.-G. (2011). A software framework based on a conceptual unified model for evolutionary multiobjective optimization: Paradiseo-moeo. *European Journal of Operational Research*, 209(2):104 – 112.

Liskov, B. (1987). Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34. ISSN 0362-1340. URL https://doi.org/10.1145/62139.62141.

Lopes Silva, M. A., de Souza, S. R., Freitas Souza, M. J., e de França Filho, M. F. (2018). Hybrid metaheuristics and multi-agent systems for solving optimization problems: a review of frameworks and a comparative analysis. *Applied Soft Computing*, 71:433–459.

Lust, T. e Teghem, J. (2010). The multiobjective traveling salesman problem: a survey and a new approach. In *Advances in Multi-Objective Nature Inspired Computing*, p. 119–141. Springer.

Meredith, A. (2009). Smart pointers on c++11. Technical report. URL http://open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2853.pdf.

Munhoz, P. L. A., Ochi, L. S., e Souza, M. J. F. (2012). Um algoritmo baseado em iterated local search para o problema de roteamento de veículos periódico. In *Anais do XXXII Encontro Nacional de Engenharia de Produção (ENEGEP)*, volume 1, p. 1–15, Bento Gonçalves/RS.

Munhoz, P. L. A., González, P. H., dos Santos Souza, U., Ochi, L. S., Michelon, P., e de A. Drummond, L. M. (2018). General variable neighborhood search for the data mule scheduling problem. *Electronic Notes in Discrete Mathematics*, 66:71 – 78.

Nascimento Silva, J. C., Coelho, I. M., Souza, U. S., Ochi, L. S., e Coelho, V. N. (2020). Finding the maximum multi improvement on neighborhood exploration. *Optimization Letters*, p. 1–19.

Resende, M. G. e Ribeiro, C. C. (2016). *Optimization by GRASP*. Springer, New York, 1 edition.

Rios, E., Coelho, I. M., Ochi, L. S., Boeres, C., e Farias, R. (2016). A benchmark on multi improvement neighborhood search strategies in cpu/gpu systems. In *2016 International Symposium on Computer Architecture and High Performance Comp Workshops (SBAC-PADW)*, p. 49–54. IEEE.

Sousa, M. M., Ochi, L. S., Coelho, I. M., e Gonçalves, L. B. (2015). A variable neighborhood search heuristic for the traveling salesman problem with hotel selection. In *2015 Latin American Computing Conference (CLEI)*, p. 1–12. IEEE.

Souza, M. J. F., Coelho, I. M., Ribas, S., Santos, H. G., e Merschmann, L. H. C. (2010). A hybrid heuristic algorithm for the open-pit-mining operational planning problem. *European Journal of Operational Research*, 207(2):1041–1051.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*. John Wiley & Sons.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85.

Zudio, A., Costa, D., Masquio, B., Coelho, I., e Pinto, P. (2018). Brkga/vnd hybrid algorithm for the classic three-dimensional bin packing problem. *Elect. Notes in Discrete Math.*, 66:175–182.