


Clustering Solutions of Multiobjective Function Inlining Problem

Kateryna Muts  

Hamburg University of Technology, Germany

Heiko Falk  

Hamburg University of Technology, Germany

Abstract

Hard real time-systems are often small devices operating on batteries that must react within a given deadline, so they must satisfy their timing, code size, and energy consumption requirements. Since these three objectives contradict each other, compilers for real-time systems go towards multiobjective optimizations which result in sets of trade-off solutions. A system designer can use the solution sets to choose the most suitable system configuration. Evolutionary algorithms can find trade-off solutions but the solution set might be large which complicates the task of the system designer. We propose to divide the solution set into clusters, so the system designer chooses the most suitable cluster and examines a smaller subset in detail. In contrast to other clustering techniques, our method guarantees that the sizes of all clusters are less than a predefined limit. Our method clusters a set by using any existing clustering method, divides clusters with sizes exceeding the predefined size into smaller clusters, and reduces the number of clusters by merging small clusters. The method guarantees that the final clusters satisfy the size constraint. We demonstrate our approach by considering a well-known compiler-based optimization called function inlining. It substitutes function calls by the function bodies which decreases the execution time and energy consumption of a program but increases its code size.

2012 ACM Subject Classification Information systems → Clustering; Software and its engineering → Compilers; Computer systems organization → Real-time systems; Mathematics of computing → Evolutionary algorithms

Keywords and phrases Clustering, multiobjective optimization, compiler, hard real-time system

Digital Object Identifier 10.4230/OASICS.WCET.2023.4

1 Introduction

Hard real-time systems are computing systems that must react before a given deadline to avoid catastrophic consequences. The Worst-Case Execution Time (WCET) of a program is its worst possible execution time independent of input data. By minimizing WCETs, we can guarantee that hard real-time systems satisfy their timing constraints. Since many embedded systems have small memories and operate on batteries, code size and energy consumption should also be minimized.

Modern compilers offer optimizations [20] that automatically improve code quality by decreasing code size, execution time, or energy consumption. But these three objectives contradict each other, i.e. when a compiler decreases one of them, it usually increases the others. An optimization problem with conflicting objectives is called multiobjective.

To choose the most desirable trade-off between the objectives, the preferences of a system designer must be incorporated into the solution process. If the designer knows the preferences before the solution process, three main approaches exist [5]: (1) all but one of the objectives are placed into constraints; (2) all objectives are combined into a single objective; (3) a decision maker conducts the solution process in direction of the desired solution.



© Kateryna Muts and Heiko Falk;

licensed under Creative Commons License CC-BY 4.0

21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023).

Editor: Peter Wägemann; Article No. 4; pp. 4:1–4:12

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

If the designer wants to know all possible trade-offs before choosing the best one, a Pareto set is generated by, e.g. using an evolutionary algorithm [6]. A Pareto set consists of trade-off solutions for which improvement in one objective worsens at least one of the other objectives.

“The magical number seven, plus or minus two” effect [19] says that humans can handle only a limited amount of information simultaneously, so the system designer cannot examine a Pareto set that consists of dozens of solutions. To improve the decision-making process, two main approaches exist:

1. select a small number of solutions that represent the entire Pareto set [25, 14, 11];
2. divide the Pareto set into clusters of small sizes [3].

In this paper, we focus on the second approach: we cluster a solution set of a multiobjective compiler-based optimization into subsets not larger than a given size.

Many existing clustering methods generate a specified number of clusters ignoring the size of each cluster. Due to “the magical number seven, plus or minus two” effect, we want to guarantee that the sizes of clusters are less than a predefined size to simplify the task of the system designer. We propose the following procedure:

1. we cluster a solution set by using an existing clustering method,
2. if the size of a cluster exceeds the predefined size, we divide the cluster into smaller clusters such that the sizes of the new clusters satisfy the size constraint,
3. we reduce the number of clusters by merging small clusters such that new clusters satisfy the size constraint.

We demonstrate the applicability of the proposed clustering method on a well-known compiler-based optimization called function inlining. This optimization substitutes a function call by the body of the callee. It potentially decreases WCET and energy consumption but increases code size, so no single optimal solution exists that minimizes the objectives simultaneously and the problem becomes multiobjective [21].

We organized the paper as follows: Section 2 presents related work, Section 3 describes the concepts of multiobjective optimization problems and introduces the proposed clustering method, Section 4 evaluates the clustering method by applying it to compiler-based optimization called function inlining, and Section 5 gives a conclusion.

2 Related Work

Compiler-based optimizations rarely employ multiobjective methods. Lokuciejewski et al. [16] considered bi-objective problems with WCET, average-case performance, and code size as objectives when searching for optimal compiler optimization sequences. Jadhav and Falk [12] presented a bi-objective static SPM allocation with WCET and energy consumption as objectives. Muts [21] studied a multiobjective function inlining with three objectives: WCET, energy consumption, and code size. Section 4 describes the multiobjective function inlining problem, since we use it to demonstrate the applicability of the clustering method proposed in this paper.

The Pareto front is a set in the objective space that represents the objectives of a Pareto set. To the best of our knowledge, clustering has been never integrated into compiler-based optimizations but it has been used to reduce Pareto fronts of other multiobjective optimizations.

Mattson, Mullur, and Messac [18] proposed a method to reduce a Pareto front such that the reduced set represents its trade-off properties. A designer controls the set size and the degree of practically insignificant trade-offs, which defines how far two solutions should be from each other to keep both of them in the reduced set. The authors used the approach to reduce Pareto fronts of bi- and tri-objective mathematical problems and a physical truss design problem.

Catania et al. [4] aimed to reduce a Pareto front that represents the performance, power, and area of an Application Specific Instruction-set Processor (ASIP) when configuring it, e.g. by setting the size of a cache. The authors used fuzzy *c*-means to partition the Pareto front and kept one solution from each cluster in the reduced set. They used Xie-Beni index [29] to identify the number of clusters.

Ishibuchi, Pang, and Shang [11] used an expected loss function to select a representative subset of the Pareto front. The expected loss function measures the loss when one solution is chosen instead of another, so the final subset minimizes the expected loss. The authors compared the approach to hypervolume-based subset selection methods [14] by considering mathematical test problems.

Kong et al. [13] proposed a clustering-based decision-making method for the multiobjective reservoir operation problem. The authors clustered a Pareto set and its Pareto front in the decision and objective spaces, respectively, to identify solutions in high-density areas of the decision and objective spaces. They selected a compromise solution by using both clustering results.

Li, Wu, and Yang [15] used TOPSIS [30] – the technique for order performance by similarity to an ideal solution – to select solutions from a Pareto front when solving a multiobjective conceptual design problem with product assembly, manufacturing, and cost as objectives. For the TOPSIS method, a designer provides a decision matrix that contains scores of the objectives for each solution of the Pareto front. TOPSIS ranks and selects solutions based on the distances to the positive- and negative-ideal solutions. The authors demonstrated the approach by considering the conceptual design of a centrifugal compressor.

Smedberg and Bandaru [26] developed an interactive decision support system that allows decision makers to visualize Pareto fronts and to study their impact in the decision space. To visualize two-dimensional projections of solutions from high-dimensional objective spaces, the authors implemented radial coordinate visualization, *t*-distributed stochastic neighbour embedding, uniform manifold approximation and projection, scatter plots, and parallel coordinate plots. To select solutions from a Pareto front, the authors implemented reference point-, lasso- and slider-based selections. To extract knowledge about the decision vectors of the selected solutions, the authors implemented two data mining techniques: Flexible Pattern Mining [2] and Simulation-Based Innovization [9] which generate decision rules. To visualize the extracted knowledge, the authors implemented a graph-based technique, where nodes represent decision rules and edges connect the rules such that the combined rule meets the significance thresholds set by a user. They used the system to study benchmark optimization problems with up to 10 objectives and real-world problems with up to six objectives.

The approaches described above extract a subset of a Pareto front that represents the entire Pareto front, and a system designer chooses the final solution from the extracted set. Such approaches might discard solutions that suit most of the system designer's requirements. Our method divides a Pareto front into clusters without discarding any solutions, and the system designer selects first a cluster and then the most suitable solution from the cluster.

Bejarano, Espitia, and Montenegro [3] studied *k*-means and *c*-means fuzzy algorithms in terms of clustering Pareto fronts obtained by solving eight artificial multiobjective problems. The authors clustered the Pareto fronts into 2–6 clusters. Both clustering algorithms produced similar clusters for continuous Pareto fronts but complementary clusters for discontinuous fronts. *K*-Means was slower than fuzzy *c*-means on the considered Pareto fronts. In contrast to our approach, this method ignores the sizes of clusters.

3 Clustering Pareto Front

Many real-world optimization problems deal with conflicting objectives, i.e. when we improve one objective, we degrade the others. Optimization problems with conflicting objectives are called multiobjective. A general *multiobjective minimization problem without constraints* is formulated as follows:

$$\text{minimize } \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})), \quad (1)$$

where $m \in \mathbb{N}$ and $f_i : X \rightarrow \mathbb{R}$ with $X \subset \mathbb{R}^d$ and $i = 1, 2, \dots, m$.

For a minimization problem with m objectives, a decision vector \mathbf{x}_1 *dominates* another vector \mathbf{x}_2 or in symbols $\mathbf{x}_1 \prec \mathbf{x}_2$, if $f_i(\mathbf{x}_1) \leq f_i(\mathbf{x}_2)$ for all $i \in \{1, 2, \dots, m\}$ and there exists $j \in \{1, 2, \dots, m\}$ such that $f_j(\mathbf{x}_1) < f_j(\mathbf{x}_2)$. A solution $\mathbf{x} \in X$ is called *Pareto optimal*, if it is not dominated by any other solution. Any multiobjective optimization problem results in a solution set P called *Pareto set* which consists of trade-off solutions: $P = \{\mathbf{x} \in X : \mathbf{x} \text{ is Pareto optimal}\} \subset \mathbb{R}^d$. The *Pareto front* F represents the subset of the objective space corresponding to the Pareto set: $F = \{\mathbf{f}(\mathbf{x}) : \mathbf{x} \in P\} \subset \mathbb{R}^m$.

Two main approaches exist to solve multiobjective problems: (1) iterative scalarization methods iteratively change their parameter values to generate a Pareto set; (2) evolutionary algorithms evolve a set of solutions - called population - over several iterations - called generations - by using bio-inspired genetic operators. Evolutionary algorithms are commonly used in practice [7], since scalarization methods produce one solution per iteration, whereas evolutionary algorithms generate a set of solutions in each iteration.

After solving a multiobjective optimization problem, a system designer selects a desirable solution from the resulting Pareto front. Since Pareto fronts are usually large and humans can handle only a limited amount of information, clustering techniques simplify the designer's task by dividing the Pareto front into clusters.

Well-known clustering methods, e.g. k-means, generate a specified number of clusters but ignore cluster sizes. We aim to cluster a Pareto front such that the size of each cluster does not exceed a given maximum size.

■ Algorithm 1 Clustering.

Require: Set $S \subset \mathbb{R}^m$, clustering algorithm *Cluster*, maximum cluster size τ , maximum distance $dist_{max}$ between two clusters to be merged

Ensure: Set of clusters $\mathcal{S} = \{S_i\}$ with $S_i \subset S$, $\bigcup S_i = S$, and $S_i \cap S_j = \emptyset$ for $i \neq j$

1: $n \leftarrow \lceil \frac{|S|}{\tau} \rceil$ ▷ Number of clusters

2: $\mathcal{S} \leftarrow \text{Cluster}(S, n)$

3: $\mathcal{S} \leftarrow \text{RefineClusters}(\mathcal{S}, \text{Cluster}, \tau)$ ▷ Algorithm 2

4: $\mathcal{S} \leftarrow \text{MergeClusters}(\mathcal{S}, \tau, dist_{max})$ ▷ Algorithm 3

Algorithm 1 presents the proposed clustering procedure. The algorithm takes a set S (Pareto front) to be clustered, a clustering algorithm *Cluster*, e.g. k-means, a desired maximum cluster size τ , and a maximum possible distance between two clusters $dist_{max}$ for merging the clusters. The parameter $dist_{max}$ guarantees that two clusters are merged only if they are close enough to each other. The algorithm returns a set of clusters \mathcal{S} . Since many clustering algorithms require a number of clusters as an input, at Line 1, the algorithm computes the number of clusters n based on the size of the set S denoted by $|S|$ and the maximum cluster size τ . The clustering algorithm *Cluster* clusters the set into n clusters. Since the clustering algorithm might generate clusters larger than the maximum size τ ,

at Line 3, the algorithm refines the clusters such that the size of each cluster is not larger than τ . The refinement might generate small clusters with sizes much smaller than τ , so at Line 4, the algorithm merges the small clusters if it is possible.

■ **Algorithm 2** Refine clusters.

Require: Set of clusters $\mathcal{S} = \{S_i\}$ with $S_i \subset \mathbb{R}^m$, clustering algorithm $Cluster$, maximum cluster size τ

Ensure: Set of refined clusters $\mathcal{R} = \{R_i\}$ with $R_i \subset \mathbb{R}^m, |R_i| \leq \tau$, and $\bigcup R_i = \bigcup S_i$

```

1:  $C_{largest} \leftarrow LargestCluster(\mathcal{S})$ 
2:  $\mathcal{R} \leftarrow \mathcal{S}$ 
3: while  $|C_{largest}| > \tau$  do
4:    $C_{refined} \leftarrow Cluster\left(C_{largest}, \left\lceil \frac{|C_{largest}|}{\tau} \right\rceil\right)$ 
5:    $\mathcal{R} \leftarrow \mathcal{R} \setminus \{C_{largest}\} \cup C_{refined}$  ▷ Update  $\mathcal{R}$ 
6:    $C_{largest} \leftarrow LargestCluster(\mathcal{R})$ 
7: end while

```

Algorithm 2 presents a procedure to refine clusters with sizes larger than τ . It takes a set of clusters \mathcal{S} , a clustering algorithm $Cluster$, and a maximum cluster size τ . The algorithm returns a set of refined clusters \mathcal{R} with sizes of all clusters less than or equal to τ .

At Lines 1 and 2, we denote by $C_{largest}$ the largest cluster in \mathcal{S} and assign the resulting set \mathcal{R} to the original set \mathcal{S} . While the size of the largest cluster $|C_{largest}|$ is larger than the desired size τ , we cluster it in smaller clusters at Line 4. We update the current set \mathcal{R} with the new clusters at Line 5 and get the largest cluster $C_{largest}$ of the updated set \mathcal{R} at Line 6. We repeat the refinement until the sizes of all clusters are less than or equal to τ .

► **Lemma 1.** *If the input set \mathcal{S} and its clusters $S_i \in \mathcal{S}$ are finite, and $0 < \tau < \infty$, Algorithm 2 terminates and returns clusters of size less than or equal to τ .*

Proof. If $|S_i| \leq \tau$ for all $S_i \in \mathcal{S}$, the algorithm terminates and returns the original clusters S_i .

To prove that the algorithm terminates if there exist clusters with sizes greater than τ , we prove that the *while* loop at Lines 3–7 terminates. We denote by \mathcal{W} a set of all clusters of size greater than τ in \mathcal{S} :

$$\mathcal{W} = \{W \in \mathcal{S} : |W| > \tau\} . \quad (2)$$

Since the input set \mathcal{S} is finite, the set \mathcal{W} is finite.

At the first iteration of the *while* loop, $C_{largest} \in \mathcal{W}$ is split into $n = \left\lceil \frac{|C_{largest}|}{\tau} \right\rceil < \infty$ clusters at Line 4. We prove by contradiction that the size of at least one of the newly created clusters is less than or equal to τ : if $|C_k^{new}| > \tau$ for all newly created clusters C_k^{new} with $k = 1, 2, \dots, n$, then

$$|C_{largest}| = \sum_{k=1}^n |C_k^{new}| > \sum_{k=1}^n \tau = n \cdot \tau = \left\lceil \frac{|C_{largest}|}{\tau} \right\rceil \cdot \tau \geq |C_{largest}|. \quad (3)$$

The newly created clusters with sizes less than or equal to τ are removed from \mathcal{W} : $\mathcal{W} = \mathcal{W} \setminus \{C_k^{new} : |C_k^{new}| \leq \tau\}$. At the next iteration, the largest cluster from the updated set \mathcal{W} is split into smaller clusters, and clusters with sizes less than or equal to τ are removed from \mathcal{W} . Since the set \mathcal{W} is finite and monotonically decreases in cardinality, we continue until the set \mathcal{W} is empty. It proves that the algorithm terminates.

The set \mathcal{R} is updated at each iteration and

$$\mathcal{R} = \{W \in \mathcal{S} : |W| \leq \tau\} \cup \{W \in \mathcal{S} : |W| > \tau\} = \{W \in \mathcal{S} : |W| \leq \tau\} \cup \mathcal{W} \quad (4)$$

Since the algorithm terminates when $\mathcal{W} = \emptyset$, the final set \mathcal{R} contains clusters of size less than or equal to τ . ◀

■ **Algorithm 3** Merge clusters.

Require: Maximum cluster size τ , set of clusters $\mathcal{S} = \{S_i\}$ with $S_i \subset \mathbb{R}^m$ and $|S_i| < \tau$, maximum distance $dist_{max}$ between two clusters to be merged

Ensure: Set of clusters $\mathcal{R} = \{R_i\}$ with $R_i \subset \mathbb{R}^m$, $|R_i| \leq \tau$, and $\bigcup R_i = \bigcup S_i$

```

1:  $\mathcal{R} \leftarrow \mathcal{S}$ 
2: for  $C \in \mathcal{R}$  do
3:    $C_{closest} \leftarrow ClosestCluster(C, \mathcal{R})$  ▷  $C_{closest} \neq C$ .
4:   if  $dist(C, C_{closest}) < dist_{max}$  AND  $|C| + |C_{closest}| < \tau$  then
5:      $C_{merged} \leftarrow Merge(C, C_{closest})$ 
6:      $\mathcal{R} \leftarrow \mathcal{R} \setminus \{C, C_{closest}\} \cup \{C_{merged}\}$  ▷ Update  $\mathcal{R}$ 
7:   go to 2
8: end if
9: end for

```

Algorithm 3 presents a procedure to merge small clusters after refinement. It takes a maximum cluster size τ to preserve the desired sizes of clusters, a set of clusters \mathcal{S} , and a maximum distance $dist_{max}$ between two clusters to be merged. The algorithm returns a set of clusters \mathcal{R} with a reduced number of clusters. At Line 1, the algorithm assigns the output set \mathcal{R} to the original set \mathcal{S} . For each cluster, it gets the cluster $C_{closest} \neq C$ closest to the current cluster C . At Line 4, we denote by $dist$ the distance between two sets $P, Q \subset \mathbb{R}^m$ defined as the Euclidean distance between the centroids \mathbf{g}_P and \mathbf{g}_Q of the sets P and Q , respectively. If the distance between the clusters is smaller than $dist_{max}$ and the sum of the cluster sizes is less than the maximum cluster size τ , we merge the clusters and update the current set \mathcal{R} . When the set \mathcal{R} is updated, the *for* loop iterates over the updated set \mathcal{R} .

► **Lemma 2.** *If $0 < \tau < \infty$, the input set \mathcal{S} is finite, and $|S_i| < \tau$ for all $S_i \in \mathcal{S}$, Algorithm 3 terminates and returns clusters of size less than or equal to τ .*

Proof. To prove that the algorithm terminates, we prove that the *for* loop terminates. The algorithm starts with \mathcal{S} assigned to \mathcal{R} . Since \mathcal{S} is finite, \mathcal{R} is finite and $|R| < \tau$ for all $R \in \mathcal{R}$.

Case 1. If for all $C \in \mathcal{R}$, the closest cluster $C_{closest}$ does not satisfy the conditions at Line 4, the loop terminates after $|\mathcal{R}| < \infty$ iterations.

Case 2. If there exists $C \in \mathcal{R}$ such that its closest cluster $C_{closest}$ satisfies the conditions at Line 4, the clusters C and $C_{closest}$ are merged, the size of the merged cluster is less than τ , and the set \mathcal{R} is updated by substituting the clusters C and $C_{closest}$ by one merged cluster. We denote by \mathcal{R}_1 the updated set \mathcal{R} . The new set \mathcal{R}_1 satisfies the following conditions:

- $|\mathcal{R}_1| = |\mathcal{R}| - 1 < \infty$,
- $|R| < \tau$ for all $R \in \mathcal{R}_1$.

The set \mathcal{R}_1 satisfies either *Case 1* or *Case 2* with $\mathcal{R} = \mathcal{R}_1$. *Case 1* terminates the algorithm and *Case 2* generates a new set of clusters which we denote by \mathcal{R}_2 . By repeating the procedure, we generate a sequence of finite sets $\{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_s\}$. The sets monotonically decrease in cardinality and $|R| < \tau$ for all $R \in \mathcal{R}_i, i = 1, 2, \dots, s$, which proves the lemma. ◀

We considered one of the most widely used clustering algorithm k-means [17] as a clustering method in Algorithms 1 and 2. We also compared the results of k-means with the results of two other clustering methods: spectral clustering [8, 22] and agglomerative clustering with the complete linkage criterion [27].

4 Results

We demonstrate the applicability of the proposed clustering method on a well-known compiler-based optimization called *function inlining* [20].

Performing function inlining, a compiler replaces a function call with the body of the function: it stores the inputs of the function call to local variables, removes the function call, inserts the function body into the code, and removes the return instruction.

Function inlining can decrease WCET and energy consumption since it

- removes call and return instructions which smooths pipeline behaviour;
- reduces parameter handling;
- enables more possibilities for subsequent optimizations, e.g. redundant path elimination or constant propagation which tightens WCET and energy consumption estimations.

Function inlining combined with other optimizations, e.g. redundant path elimination, might decrease code size but it often increases code size due to duplicated function bodies.

The multiobjective function inlining problem with WCET, energy consumption, and code size as objectives is formulated as follows [21]:

- *decision space*: $X = [0, 1]^d$, where the dimension d is equal to the total number of function calls in a program. A decision vector $\mathbf{x} = (x_1, x_2, \dots, x_d)$ is defined as follows:

$$x_i = \begin{cases} 1, & \text{the function is inlined at the function call } i, \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

- *objective function*: $\mathbf{f} = (\text{WCET}, \text{Energy Consumption}, \text{Code Size})$
- *optimization problem*: $\min_{\mathbf{x} \in X} \mathbf{f}(\mathbf{x})$.

After solving the multiobjective function inlining problem by the evolutionary algorithm MBPOA [28] and getting a Pareto front, we cluster the solutions as described in the previous section.

We use the WCC compiler framework [10] for the ARM Cortex-M0 microcontroller. We computed WCET and energy consumption by AbsInt's aiT and EnergyAnalyser 20.10i [1] and code size by WCC. We ran all evaluations on a computer with Dual CPU Intel Core i7-5600U, RAM 15 GB, 2 CPU cores, and 2.60 GHz CPU frequency. We implemented Algorithms 1, 2, and 3 in Python 3.10.

Function inlining shows the most significant improvement of a final executable in combination with other optimizations, e.g. constant propagation and dead code elimination, so we perform all evaluations with the compiler optimization level `O2`. The Cortex-M0 microcontroller lacks a hardware floating-point unit, so we use the WCC software math library to tackle this issue. We considered functions of the floating-point library as candidates for inlining. We used benchmarks from EEMBC benchmark suite [24] where MBPOA resulted in more than 10 solutions because otherwise, the clustering problem is trivial. We assumed that a benchmark fits into the Flash memory of the architecture.

When clustering a Pareto front found by MBPOA, we consider only meaningful solutions, i.e. solutions with decreasing WCET or energy consumption compared to the original program. Table 1 lists the total number of solutions and the number of meaningful solutions. For all benchmarks, except `bitmnp01`, the applied constraint insignificantly reduced the Pareto fronts.

■ **Table 1** The number of solutions when solving the multiobjective function inlining problem by evolutionary algorithm MBPOA.

	Benchmark												
Number of solutions	a2time01	aifirf01	basefp01	bitmnp01	cacheb01	canrdr01	des	iirfft01	pntrch01	puwmod01	rspeed01	tblock01	ttsprk01
Total	29	38	13	41	21	42	15	26	43	91	17	13	96
Meaningful solutions	29	37	13	11	20	41	15	26	43	91	16	13	96

We used Algorithm 1 to cluster the Pareto fronts. We utilized k-means, spectral clustering, and agglomerative clustering as input clustering methods in the algorithm. We used the implementation of the clustering methods provided by the tool `scikit-learn` [23]. We preserved the `scikit`'s default values provided for the methods. We set maximum cluster size $\tau = 7$ in Algorithm 1 due to “the magical number seven, plus or minus two” effect [19], and maximum distance $dist_{max}$ was computed as described in the following remark:

► **Remark 3.** We did not pass maximum distance $dist_{max}$ as input to Algorithms 1 and 3 but computed it in Algorithm 3 as follows:

$$dist_{max} = \frac{d_{max}}{n - 1} , \quad (6)$$

where n is the number of clusters in the input set \mathcal{S} and d_{max} is the maximum distance between two points from the union of sets $S_i \in \mathcal{S}$:

$$d_{max} = \max_{p, q \in \cup S_i} \|p - q\| . \quad (7)$$

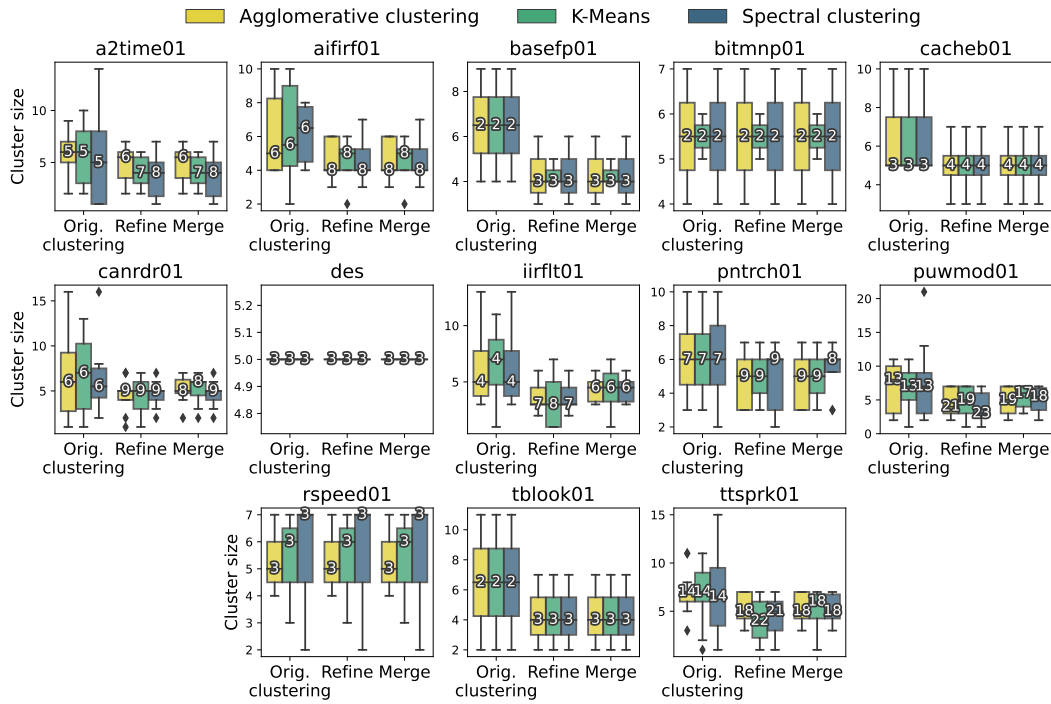
Figure 1 shows cluster sizes as box plots after each stage of Algorithm 1 shown in x-axes: original clustering, refinement of large clusters, and merger of small clusters. The figure shows the results when using agglomerative clustering, k-means, and spectral clustering in Algorithm 1. The numbers on the medians show the total number of clusters. E.g. for the benchmark `canrdr01`, the original agglomerative clustering resulted in six clusters with the largest cluster size 16; after refinement, the algorithm produced 9 clusters with the maximum cluster size 7; and after merging, the algorithm returned 8 clusters. As expected, refinement reduces the cluster sizes to the desired value but increases the number of clusters, whereas merging small clusters may reduce the number of clusters.

If we choose the best clustering method to be used in Algorithm 1 based on the number of final clusters: fewer clusters are better, k-means outperforms the two other clustering methods. For all benchmarks, Algorithm 1 with k-means resulted in the same or smaller number of clusters than when it was combined with the two other clustering methods.

Table 1 shows that six benchmarks `bitmnp01`, `des`, `rspeed`, `basefp01`, `cacheb01`, and `tblock` have a few meaningful solutions compared to the remaining benchmarks. For these benchmarks, Figure 1 shows that

- the refinement and merging stages of Algorithm 1 preserved the original clusters (`bitmnp01`, `des`, `rspeed`) or
- refinement was required to generate clusters of the desired sizes (`basefp01`, `cacheb01`, and `tblock`), whereas merging was useless.

For the remaining seven benchmarks with more solutions to be clustered, the refinement stage was necessary to generate clusters of the desired sizes. For all these benchmarks, except `aifirf01`, the merging stage reduced the number of final clusters for at least one of the



■ **Figure 1** Cluster sizes after each stage of Algorithm 1 when using agglomerative, k-means, and spectral clustering. The numbers on the medians show the total number of clusters.

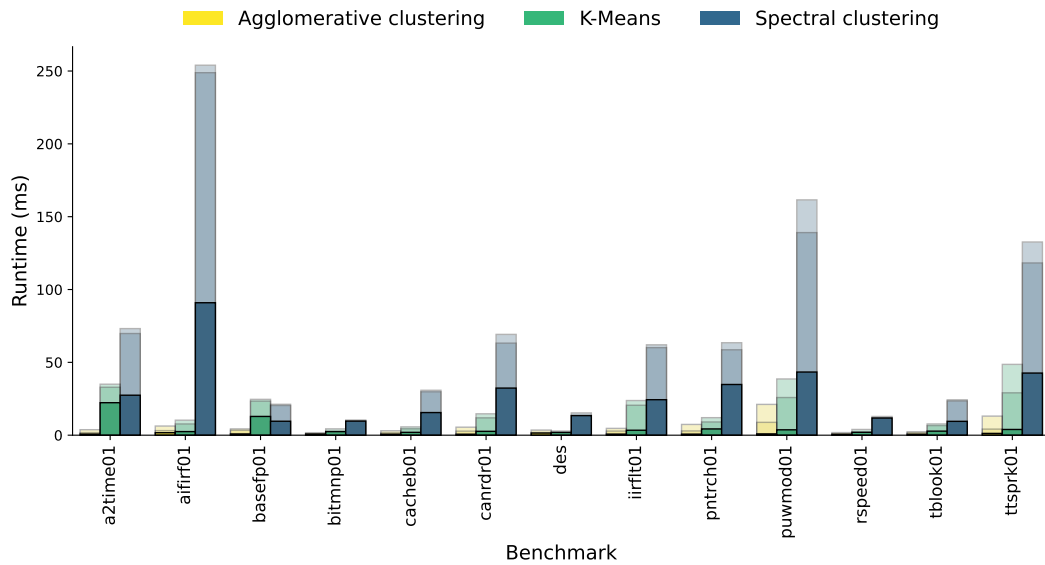
considered clustering methods. We observed the most significant reduction for benchmark `puwmod01` with spectral clustering where the merge algorithm returned 18 clusters instead of 23 produced at the refinement stage. This benchmark is one of the largest benchmarks considered in the evaluation.

► **Remark 4.** For large Pareto fronts, Algorithm 1 results in many clusters when the maximum cluster size τ is small. We observed these results for benchmarks `puwmod01` and `ttsprk01`. If a Pareto front is large, a system designer should iteratively invoke Algorithm 1:

1. set τ to a large value to divide the Pareto front into fewer clusters of large sizes;
 2. choose the best cluster;
 3. pass the cluster to the algorithm and decrease τ to divide the cluster into smaller clusters.
- Repeat Steps 2 and 3 until the desired maximum cluster size is achieved.

Figure 2 shows the runtime of the stages of Algorithm 1 as a stack diagram with three parts starting from 0: original clustering, refinement, and merging. For all benchmarks, the runtime of the three approaches was less than 0.3s. Agglomerative clustering resulted in the shortest runtime for all benchmarks, except `des`. For 12 out of 13 benchmarks, Algorithm 1 with spectral clustering was the slowest approach. For most benchmarks, the refinement stage was the most time-consuming part of the algorithm, whereas the merge stage was the least-time consuming part.

To sum up, in general, the three considered approaches showed very similar results according to Figure 1, but Figure 2 shows that Algorithm 1 with agglomerative clustering was finished in less than 0.03s for all benchmarks. It is 0.012s and 0.066s faster, on average, than Algorithm 1 with k-means and spectral clustering, respectively.



■ **Figure 2** Runtime of the stages of Algorithm 1 with agglomerative clustering, k-means, and spectral clustering. The parts of the stack diagram starting from 0: original clustering, refinement, and merging.

5 Conclusion

We presented a method to cluster trade-off solutions of a multiobjective optimization problem. To guarantee that the size of all clusters is less than a predefined limit, our method clusters the solutions by using a known clustering method, refines clusters with exceeding sizes, and merges small clusters if possible. The last step tries to reduce the number of clusters which may increase after the refinement.

We demonstrated our approach by clustering solutions of multiobjective function inlining problem. We compared the results by using three clustering methods as a base for our approach: k-means, agglomerative and spectral clusterings. By using the three clustering methods, the proposed clustering techniques showed similar results in terms of the number of clusters and their sizes, but it showed the smallest runtime when using agglomerative clustering.

In future work, the proposed method should be verified by using other multiobjective optimizations, including compiler-based optimizations.

References

- 1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers. <https://www.absint.com/ait/index.htm>, 2022.
- 2 Sunith Bandaru, Amos H.C. Ng, and Kalyanmoy Deb. Data mining methods for knowledge discovery in multi-objective optimization: Part B - New developments and applications. *Expert Systems with Applications*, 70:119–138, March 2017. doi:10.1016/j.eswa.2016.10.016.
- 3 Lilian Astrid Bejarano, Helbert Eduardo Espitia, and Carlos Enrique Montenegro. Clustering Analysis for the Pareto Optimal Front in Multi-Objective Optimization. *Computation*, 10(3):37, March 2022. doi:10.3390/computation10030037.
- 4 Vincenzo Catania, Giuseppe Ascia, Maurizio Palesi, Davide Patti, and Alessandro G. Di Nuovo. Fuzzy decision making in embedded system design. In *Proceedings of the 4th International*

- Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 223–228, October 2006. doi:10.1145/1176254.1176309.
- 5 Carlos A. Coello, David A. Van Veldhuizen, and Gary B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer-Verlag GmbH, October 2007.
 - 6 Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, New York, NY, USA, 2001.
 - 7 Kalyanmoy Deb. Multi-objective Optimization. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 403–449. Springer US, Boston, MA, 2014. doi:10.1007/978-1-4614-6940-7_15.
 - 8 Wilm E. Donath and Albert J. Hoffman. Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.
 - 9 Catarina Dudas, Amos H.C. Ng, and Henrik Boström. Post-analysis of multi-objective optimization solutions using decision trees. *Intelligent Data Analysis*, 19(2):259–278, April 2015. doi:10.3233/IDA-150716.
 - 10 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, July 2010. doi:10.1007/s11241-010-9101-x.
 - 11 Hisao Ishibuchi, Lie Meng Pang, and Ke Shang. Solution Subset Selection for Final Decision Making in Evolutionary Multi-Objective Optimization, June 2020. doi:10.48550/arXiv.2006.08156.
 - 12 Shashank Jadhav and Heiko Falk. Multi-Objective Optimization for the Compiler of Real-Time Systems based on Flower Pollination Algorithm. In *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 45–48. ACM, May 2019. doi:10.1145/3323439.3323977.
 - 13 Yanjun Kong, Yadong Mei, Xianxun Wang, and Yue Ben. Solution Selection from a Pareto Optimal Set of Multi-Objective Reservoir Operation via Clustering Operation Processes and Objective Values. *Water*, 13(8):1046, January 2021. doi:10.3390/w13081046.
 - 14 Tobias Kuhn, Carlos M. Fonseca, Luís Paquete, Stefan Ruzika, Miguel M. Duarte, and José Rui Figueira. Hypervolume Subset Selection in Two Dimensions: Formulations and Algorithms. *Evolutionary Computation*, 24(3):411–425, 2016. doi:10.1162/EVCO_a_00157.
 - 15 Congdong Li, Run Wu, and Weiming Yang. Optimization and selection of the multi-objective conceptual design scheme for considering product assembly, manufacturing and cost. *SN Applied Sciences*, 4(4):91, March 2022. doi:10.1007/s42452-022-04973-6.
 - 16 Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Approximating Pareto optimal compiler optimization sequences – a trade-off between WCET, ACET and code size. *Software: Practice and Experience*, 41(12):1437–1458, May 2011. doi:10.1002/spe.1079.
 - 17 James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, volume 5.1, pages 281–298. University of California Press, January 1967.
 - 18 Christopher Mattson, Anoop Mullur, and Achille Messac. Minimal Representation of Multiobjective Design Space Using a Smart Pareto Filter. In *9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Multidisciplinary Analysis Optimization Conferences. American Institute of Aeronautics and Astronautics, September 2002. doi:10.2514/6.2002-5458.
 - 19 George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956. doi:10.1037/h0043158.
 - 20 Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1998.
 - 21 Kateryna Muts. *Multiobjective Compiler-Based Optimizations for Hard Real-Time Systems*. PhD thesis, TUHH Universitätsbibliothek, December 2022. doi:10.15480/882.4799.

- 22 Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 849–856, Cambridge, MA, USA, January 2001. MIT Press.
- 23 Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, January 2012. [arXiv:1201.0490v4](https://arxiv.org/abs/1201.0490v4).
- 24 Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro*, 29(5):18–29, September 2009. doi:10.1109/MM.2009.74.
- 25 Mike Preuss and Simon Wessing. Measuring Multimodal Optimization Solution Sets with a View to Multiobjective Techniques. In *EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation IV*, Advances in Intelligent Systems and Computing, pages 123–137, Heidelberg, 2013. Springer International Publishing. doi:10.1007/978-3-319-01128-8_9.
- 26 Henrik Smedberg and Sunith Bandaru. Interactive knowledge discovery and knowledge visualization for decision support in multi-objective optimization. *European Journal of Operational Research*, 306(3):1311–1329, May 2023. doi:10.1016/j.ejor.2022.09.008.
- 27 Abdulhamit Subasi. *Practical Machine Learning for Data Analysis Using Python*. Academic Press, London [England]; San Diego, CA, 2020.
- 28 Ling Wang, Haoqi Ni, Weifeng Zhou, Panos M. Pardalos, Jiating Fang, and Minrui Fei. MBPOA-based LQR controller and its application to the double-parallel inverted pendulum system. *Engineering Applications of Artificial Intelligence*, 36:262–268, November 2014. doi:10.1016/j.engappai.2014.07.023.
- 29 Xuanli L. Xie and Gerardo Beni. A validity measure for fuzzy clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):841–847, 1991. doi:10.1109/34.85677.
- 30 Zhongliang Yue. An extended TOPSIS for determining weights of decision makers with interval numbers. *Knowledge-Based Systems*, 24(1):146–153, February 2011. doi:10.1016/j.knsys.2010.07.014.