



Towards Multi-Objective Dynamic SPM Allocation

Shashank Jadhav  

Hamburg University of Technology, Germany

Heiko Falk  

Hamburg University of Technology, Germany

Abstract

Most real-time embedded systems are required to fulfill timing constraints while adhering to a limited energy budget. Small ScratchPad Memory (SPM) poses a common hardware constraint on embedded systems. Static SPM allocation techniques are limited by the SPM's stringent size constraint, which is why this paper proposes a Dynamic SPM Allocation (DSA) model at the compiler level for the dynamic allocation of a program to SPM during runtime. To minimize Worst-Case Execution Time (WCET) and energy objectives, we propose a multi-objective DSA-based optimization. Static SPM allocations might inherently use SPM sub-optimally, while all proposed DSA optimizations are only single-objective. Therefore, this paper is the first step towards a DSA that trades WCET and energy objectives simultaneously. Even with extra DSA overheads, our approach provides better quality solutions than the state-of-the-art multi-objective static SPM allocation and ILP-based single-objective DSA approach.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Compilers; Mathematics of computing → Discrete mathematics

Keywords and phrases Multi-objective optimization, Embedded systems, Compilers, Dynamic SPM allocation, Metaheuristic algorithms

Digital Object Identifier 10.4230/OASICS.WCET.2023.6

Funding This work is part of a project that received funding from NXP Semiconductors.

1 Introduction

Real-time embedded systems must satisfy hard timing constraints and often operate on a limited energy budget. To optimize such systems, it is important to consider WCET and energy consumption of the program. As SPMs are fast and energy-efficient local memories, various static SPM allocation-based optimizations have been explored to exploit their potential. But, their small size gravely constrains the static optimization problem. Therefore, we propose a compiler-level DSA model in this paper to exploit the memory subsystem and circumvent the SPM size constraint. Additionally, for the very first time, we propose a strategy to perform WCET and energy analyses of such dynamically allocated programs statically at compile-time, enabling us to perform DSA-based multi-objective optimization during compilation.

DSA is traditionally an important task for Operating Systems (OS), but the execution times of OS-based allocation techniques are difficult to predict and guarantee. The compiler-based DSA has been investigated before for reasonably limited architectures, and only single-objective optimizations to minimize either WCET or energy have been considered. However, the program's WCET- and energy-critical areas may differ, and optimizing for WCET alone can negatively impact energy consumption and vice versa. Therefore, in this paper, we propose for the very first time a multi-objective optimization that uses the proposed DSA model and simultaneously optimizes the WCET and energy consumption.

We implemented the proposed DSA-based multi-objective optimization within the WCET-aware C Compiler (WCC) [6] framework and solved using two metaheuristic algorithms, namely Flower Pollination Algorithm (FPA) [25] and Strength Pareto Evolutionary Algorithm



© Shashank Jadhav 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. 6; pp. 6:1–6:12

OpenAccess Series in Informatics



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

(SPEA) [26]. For the sake of brevity, we refer to this optimization run as MO_D in this paper. Furthermore, we compare the evaluation results of MO_D with a static SPM allocation-based multi-objective optimization and an ILP-based single-objective DSA optimization referred to as MO_S and SO_D , respectively, in this paper. The key contributions of this paper are:

- For the very first time, we formulated a DSA model that allocates memory objects from Flash to SPM during runtime and supports both WCET and energy analyses.
- We proposed a MO_D that uses metaheuristic algorithms to solve the said problem.
- MO_D is compared with MO_S and SO_D using real-world benchmark suites from EEMBC.
- DSA introduces significant WCET and energy overheads still, the quality of solutions from MO_D is slightly better than the ones from MO_S , and MO_D outperforms SO_D .

The paper is organized as follows: Sec. (2) provides an overview of the related work. Sec. (3) discusses the proposed DSA model. Sec. (4) presents MO_D , and Sec. (5) presents the evaluation results. A conclusion and discussion of the future work conclude the paper.

2 Related Work

DSA facilitates copying code and data objects on and off memories during runtime and exploits the memory subsystem to its fullest potential [13]. DSA-based approaches proposed in the past are in the context of single-objective optimizations [22, 19, 7, 20]. Deverge et al. [5] proposed DSA for static and stack data to minimize WCET. Kim et al. [14] proposed dynamic instructions allocation at the function level for minimizing WCET using direct memory access transfers. Verma et al. [24] proposed an SPM overlay approach for data and instruction allocation that minimizes the energy consumption of the program. Liu et al. [15] built upon Verma’s scratchpad overlay model by considering a multi-level SPM architecture for multi-core processors for minimizing WCET. These approaches consider the DSA of either code or data and focus on only single objective optimization.

Performing compiler-level multi-objective optimizations has rarely been exploited. Lokuciejewski et al. [16] proposed a stochastic evolutionary approach to find Pareto optimal compiler optimization sequences. He considered trade-offs between Average-Case Execution Time (ACET) and WCET, as well as WCET and code size. Muts et al. [17] proposed a function-inlining-based multi-criteria optimization that traded WCET, energy, and code size. Hoste et al. [10] proposed a multi-objective optimization framework that used evolutionary algorithms to explore compiler optimization levels and automatically finds Pareto-optimal optimization levels. In the past, we proposed a multi-objective optimization using FPA to perform compiler-level static SPM allocation [11]. However, none of these multi-objective optimizations focus on dynamic allocation to exploit the memory subsystems. Therefore, this paper is the first step toward performing DSA-based multi-objective optimization that simultaneously minimizes the WCET and energy consumption of the program.

3 Dynamic SPM Allocation Model

In this section, we propose a DSA model within WCC that can dynamically allocate memory objects at runtime. DSA is the process of allocating memory objects dynamically during the runtime of a program. Performing compiler-level DSA allows us to predetermine the WCET- and energy-intensive memory object that could be dynamically copied to SPM during runtime such that WCET and energy objectives are minimized.

3.1 Memory Objects

A memory object (*memObj*) is defined as the finest granularity program fragment considered in a DSA problem. A Basic Block (BB), a code sequence with no branches except possibly at the exit, can be considered a *memObj*. But, while performing DSA, the re-usability of a *memObj* plays a critical role. Executing a *memObj* only once from SPM leads to certain WCET or energy reductions, but the overheads for dynamically copying such *memObj* can overshadow these savings. For this reason, this paper considers only *memObjs* that are executed several times, i.e., that exhibit a high re-usability. By construction, such *memObjs* are loops and functions. Therefore, we consider functions and loops as code memory objects (*memObj_c*) for dynamic allocation. Loops and functions can provide the re-usability of BBs and reduce additional overhead introduced by the movement of individual BBs.

A global data variable is *live* throughout the complete execution of the code. The dynamic allocation of global data variables can introduce unnecessary overheads in terms of WCET and energy consumption. Therefore, global data variables (*memObj_d*) are allocated statically either to SPM or to Flash by our approach. On the other hand, the scope of local data variables only exists within some parts of a single function. Therefore, our approach considers local data variables as part of the functions or loops within which they are being used and are dynamically allocated in conjunction with them.

The underlying exemplary architecture considered while modeling the DSA model consists of the Flash, an instruction SPM (*ISPM*), and a data SPM (*DSPM*). Let $\mathcal{M} \subset \mathcal{F} \cup \mathcal{L} \cup \mathcal{G}$ be a set of *memObj* that is a union of the set of functions \mathcal{F} , the set of loops \mathcal{L} , and the set of global data variables \mathcal{G} within a program. Let $x \in \{0, 1\}^d$ represent a d -dimensional binary decision variable vector that describes which *memObj* is allocated in which memory. x is a block vector, where the subvector $x_{1:F} = (x_1, \dots, x_F)$ are decision variables for functions, the subvector $x_{(F+1):(F+L)} = (x_{F+1}, \dots, x_{F+L})$ are decision variables for loops, and the subvector $x_{(F+L+1):(F+L+G)} = (x_{F+L+1}, \dots, x_{F+L+G})$ are decision variables for global data variables. F is the total number of functions, L is the total number of loops, and G is the total number of global data variables within \mathcal{M} . $d = (F + L + G)$ is the total number of *memObj*. Each coordinate $x_i, i = \overline{1, d}$ of vector x corresponds to a specific *memObj*:

$$x_i = \begin{cases} 1, & \text{if } \mathcal{M}_i \text{ is in } ISPM \\ 0, & \text{if } \mathcal{M}_i \text{ is in Flash} \end{cases} \quad x_i = \begin{cases} 1, & \text{if } \mathcal{M}_i \text{ is in } DSPM \\ 0, & \text{if } \mathcal{M}_i \text{ is in Flash} \end{cases} \quad (1)$$

$$\forall i = \overline{1, (F + L)} \quad \forall i = \overline{(F + L + 1), (F + L + G)}$$

The *memObj_c* referring to decision variables $x_i, \forall i = \overline{1, (F + L)}$ are allocated dynamically from Flash to *ISPM*, and the *memObj_d* referring to $x_i, \forall (F + L + 1), (F + L + G)$ are allocated statically from Flash to *DSPM*.

3.2 Liveness Analysis

A *memObj* is *live* at an edge $e \in E$ of the control flow graph $G(N, E)$ if there exists a back path from the edge e to a node $n \in N$, where the *memObj* is defined without being redefined at any other node along the path [24]. For the sake of brevity, a detailed explanation of the standard liveness analysis is omitted [2]. We perform liveness analysis within WCC at the function level. Each function is analyzed to determine the *live* range of a *memObj*. A function *memObj* is defined (*def*) when it is first called within the function currently under analysis. The subsequent calls for that function are categorized as *use*. The *live* range of the function *memObj* is the path, i.e., set of BBs, from *def* until the last *use*.

A loop *memObj* is defined at the live-out edge of the loop-entry BB. All BBs within the loop are categorized as *use*. The *live* range of the loop *memObj* is the loop-entry's predecessor BB, i.e., *def*, until the loop-exit BBs. In case of multiple entries and exits for a loop, all the BBs are considered part of the *live* range of the loop *memObj*. In the case of nested loops, each loop is considered an individual *memObj* entity. The loop *memObj* nested within another loop is defined at the live-out edge of the top-enclosing loop entry's BB, and the BBs within the individual loop *memObj* are categorized as *use*. The *live* range of a nested loop *memObj* spans over all the BBs contained within the top-enclosing loop. In the case of global data variables, they are considered *live* throughout the complete execution of the code.

Let $\Lambda = \{\lambda_1, \dots, \lambda_{(F+L)}\}$, where $(F+L)$ is the total number of *memObj_c* and λ_i be a set of BBs *live* for *memObj_c* M_i , i.e., $\lambda_i = \{b_p \mid \forall p \ b_p \text{ is live for } M_i\}$. Let $C = \{c_1, \dots, c_{(F+L)}\}$, where c_i be a $(F+L)$ -dimensional binary vector that determines if there exists an overlap of *live* ranges between the i^{th} *memObj_c* and others. Each coordinate $c_{ij}, j = \overline{1, (F+L)}$ corresponds to a conflict of i^{th} *memObj* with the j^{th} *memObj*, i.e.,

$$c_{ij} = \begin{cases} 1, & \text{if } \exists b_p \mid b_p \in \lambda_i \ \& \ b_p \in \lambda_j, \ \& \ i \neq j \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

When solving the address assignment problem, *liveness* conflicts between *memObjs* are considered. These conflicts prevent the allocation of *memObjs* that share conflicting *liveness* to the same memory address within SPM.

3.3 Address Assignment

DSA allows copying of *memObj_c* from Flash to SPM during runtime dynamically. But, we must be careful that any *memObj_c* is not overwritten during its execution. Therefore, we need to solve an address assignment problem at compile-time such that no two *memObj_c* that are *live* at the same time are allocated to the same memory addresses. We solve the address assignment algorithm within WCC to appropriately allocate address spaces to *memObj_c* for proper dynamic allocation. Moreover, all BBs contained within a *memObj_c* are not always placed within consecutive memory addresses. For example, a loop within a nested loop could have BBs located in consecutive memory addresses, followed by another loop, and then followed by the remaining BBs. In this case, we need two distinct memory copy functions to dynamically allocate the whole *memObj*. Therefore, we define an address object (*addrObj*) as a set of BBs from the *memObj_c* such that they are placed within consecutive memory addresses. Solving the address assignment problem provides us with the start address, the destination address, and the size of the *addrObj*, which are needed for their dynamic allocation during runtime. Let T_i be the total number of *addrObjs* associated with the *memObj_c* M_i , and \mathcal{T} be the total number of *addrObjs* that need dynamic allocation.

To solve the address assignment problem, we use a combination of the first-fit and best-fit heuristics [8]. The first-fit heuristic fit as many *addrObj* as possible within SPM until the SPM is full. If two *addrObj* are adjacent within the Flash, then we try to place them similarly in SPM. Once the SPM is full, we run the best-fit heuristic to find the best possible place in SPM for the remaining *addrObj*. *memObjs* with *liveness* conflicts are not overlapped within the memory addresses. In case the size of the *addrObj* is larger than the already placed *addrObjs*, then we try to find multiple adjacent *addrObj* that fit the considered *addrObj*. If all *addrObjs* are assigned to SPM, then the algorithm returns 0. If that is not the case, then it returns $(\mathcal{T} - \eta)$, where η is the number of *addrObjs* assigned to SPM.

Algorithm 1 Generation and Analyses of Dynamically Allocated Code.

```

1: For each solution  $x$ 
2: for  $i = 1 : (F + L + G)$  do                                ▷ For each  $\mathcal{M}_i$ 
3:   if  $x_i == 1$  then                                       ▷ If  $\mathcal{M}_i$  is placed in SPM
4:     if  $i \leq (F + L)$  then                                  ▷ For code  $\mathcal{M}_i$ 
5:       Perform jump correction
6:     else                                                    ▷ For data  $\mathcal{M}_i$ 
7:       Statically allocate global data object to  $DSPM$ 
8: Perform Address Assignment
9: if  $(\mathcal{T} - \eta) \neq 0$  then
10:   Repair solution  $x$ 
11:   Repeat Steps 2–8
12: Insert memcpy() calls and call literal pool placement algorithm
13: Again perform Address Assignment to accommodate memcpy() calls and literal pool changes
14: Generate a static version of the code and perform WCET and energy analyses
15: Collect Analyses results and discard the static version of the code

```

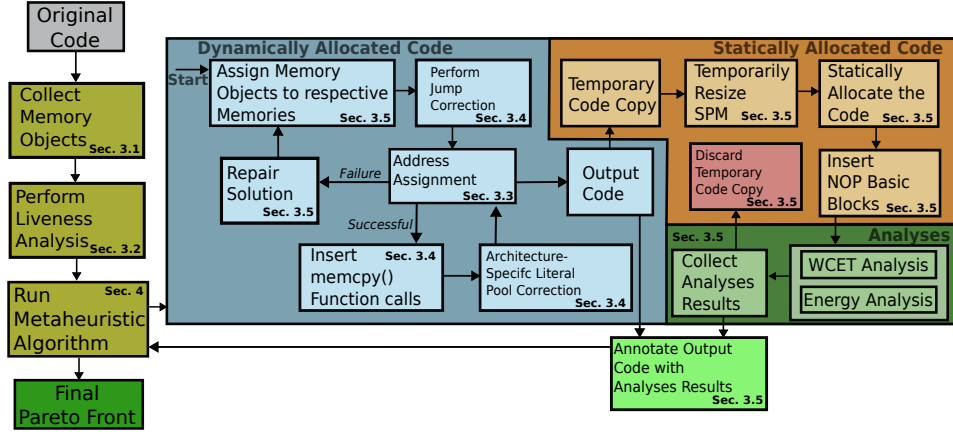
3.4 Code Transformation for Dynamic SPM Allocation

We perform code transformations during compile time to dynamically move *addrObj* from one memory to another. To copy a *addrObj* during runtime, we insert a memory copy function (*memcpy()*) at the assembly level. The *memcpy()* is allocated to the Flash, and it takes the start address (α), size (β), and destination address (δ) of an *addrObj* as inputs. The values for α , β , and δ are available after solving the address assignment problem. These three parameters enable *memcpy()* to copy code from one memory to another during runtime. After solving the address assignment problem and if $(\mathcal{T} - \eta) = 0$, we place *memcpy()* calls at the assembly level before the *memObj_c*. This call to *memcpy()* enables the program to copy code from α to δ during runtime. Once the code is copied from one memory to another during runtime, we also want our code to jump to δ instead of α . Therefore, we perform jump correction, such that previously valid jumps to α are replaced by new jumps to δ , enabling the code to jump to a proper destination address during runtime.

The code transformation for DSA is architecture-specific, i.e., we implemented these mechanisms for an ARMv7-based architecture within WCC. Placing *memcpy()* calls and jump correction code at the assembly level could increase the distance between BBs and literal pools referred to by the BBs. Therefore, we implemented an ARMv7-specific algorithm to fix literal pool placement within WCC. To fix literal pool placement, we move the literal pool near the BB referring them and generate a jump over the moved literal pool [3]. For the sake of brevity, we are skipping WCC-related implementation details and the detailed explanation of the architecture-specific code transformation for DSA. On the other hand, no major code transformation is needed for the static allocation of global data variables *memObj_d*. We assign *memObj_d* to respective memories according to their allocations at the assembly level, and an additional startup code to move *memObj_d* statically is needed.

3.5 Analyses of Dynamically Allocated Code

Performing WCET and energy analyses for dynamically allocated code at compile-time using static analysis tools is not feasible. To circumvent this problem, we generate a temporary static version of the code by virtually placing *memObj* within different memories according to the solution x . We assume that all the required *memObj* will fit within SPM, i.e., the SPM is temporarily resized to generate the static version of the code. Then, we assign *memObj*



■ **Figure 1** Dynamic SPM Allocation-based Multi-Objective Optimization Framework.

to respective memories, insert the *memcpy()* function calls at appropriate places according to the results from the address assignment algorithm, and perform jump correction. The *memcpy()* function is annotated with parametric loop bounds, which helps WCET and energy analyzers to calculate the contribution of *memcpy()* for each *memObj_c*. Therefore, WCET and energy contributions of dynamically copying *memObj_c* are collected at the compiler level.

Assigning *memObj_c* to SPM can affect the memory addresses of the remaining code in Flash. In order to keep the memory layout in Flash unchanged for static analyses, we insert *NOP* BBs in Flash in place of *memObj_c* that are statically allocated to SPM. These *NOP* BBs do not contribute to the final WCET and energy analysis results. Once the static version of the code is analyzed, we collect the results and discard this temporary code version. Algorithm (1) describes the process needed for DSA code generation and analyses. The address assignment algorithm may fail to assign all the *memObj_c* to appropriate addresses within SPM due to liveness conflicts. So, instead of discarding the whole solution, we repair the solution and then generate the dynamically allocated code for the repaired solution. We repair the solution by moving the *memObj_c* that are not assigned to an SPM address by the address assignment algorithm back to Flash.

4 Multi-Objective Dynamic SPM Allocation-based Optimization

In this section, we formulate a compiler-level DSA-based multi-objective optimization that minimizes the program’s WCET and energy consumption. Fig. (1) depicts the proposed DSA-based multi-objective optimization framework. The figure presents the flow between different aspects of DSA code generation and its analyses explained in Sec. (3). A multi-objective optimization problem performing DSA can be mathematically formulated as a minimization problem as follows.

$$\min_x F(x) = (F_1(x), F_2(x)), \quad \text{subject to} \quad x_{(F+1):(F+L)} = x_{(F+1):(F+L)} + \tau \quad (3)$$

$$(\mathcal{T} - \eta) = 0$$

where the objective function $F(x) \in \mathbb{R}^2$ represents WCET and energy consumption corresponding to a solution vector x . $x \in X$ represents a d -dimensional binary decision variable vector, where Eq. (1) describes each coordinate x_i , $i = \overline{1, d}$ and $X \subset \{0, 1\}^d$ is the search space of the DSA problem. The first constraint is applied based on the liveness analysis, i.e.,

if a function is allocated to SPM and the loops contained within that function are allocated to Flash, then it is logical that the loops contained within that function are also allocated within the SPM. Within this constraint, τ is a L -dimensional binary vector, and each element τ_l , $l = \overline{1, L}$ is '1' if l^{th} loop is in Flash and there exists a function f placed in SPM that has *liveness* conflict with the considered loop, i.e.,

$$\tau_l = \begin{cases} 1, & \text{if } x_{F+l} = 0 \ \& \ (\exists f \mid \lambda_{F+l} \subseteq \lambda_f \in \Lambda_{1:F}) \ \& \ x_f = 1 \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

where, $\lambda_{(F+l)}$ represents the set of BBs *live* for the $(F+l)^{\text{th}}$ *memObj* or l^{th} loop *memObj*, and $\Lambda_{1:F}$ is the set of sets of *live* BBs for function *memObj* (c.f. Sec. (3.2)). Furthermore, the second constraint $(\mathcal{T} - \eta) = 0$ says that the address assignment algorithm should return 0 for the solution vector x . We utilize the metaheuristic algorithms FPA and SPEA To solve the multi-objective optimization. In order to identify the trade-offs between different solutions of multi-objective optimization, we introduce a few definitions.

► **Definition 1.** Let $x_1, x_2 \in X$ and $F(x) = (F_1(x), F_2(x))$, then x_1 dominates x_2 , i.e., $x_1 \prec x_2$, if $\forall t \in \{1, 2\} F_t(x_1) \leq F_t(x_2)$ and $\exists r \in \{1, 2\} : F_r(x_1) < F_r(x_2)$.

► **Definition 2.** The solutions that are not dominated by any other solution are called Pareto optimal solutions. The set of all such Pareto optimal solutions is called Pareto optimal set, and the set of corresponding objective vectors is called the Pareto optimal front.

The initial population for FPA and SPEA are defined randomly and can influence the final Pareto front. Therefore, we perform evaluations using five different initial populations reducing the influence of the initial population on the final results. As the true Pareto front is unknown for our problem, we combine approximated Pareto fronts found by several runs of the algorithm for different initial populations into a set of nondominated points as reference Pareto front \mathcal{P} . To evaluate and compare the quality of the proposed MO_D , we use the following quality indicators:

► **Definition 3.** Coverage ($\mathcal{C} \in [0, 1]$) [26] describes the total number of dominated points in a solution set A , i.e., $\mathcal{C} = 1 - \frac{|\{a \in A : \exists p \in \mathcal{P}, a \preceq p\}|}{|A|}$

► **Definition 4.** Non-Dominated Ratio ($NDR \in [0, 1]$) [9] measures the ratio of non-dominated solutions that are contributed by a particular solution set A to the non-dominated solutions provided by all solutions sets, i.e., $NDR = \frac{|\mathcal{P} \cap A|}{|\mathcal{P}|}$

► **Definition 5.** Non-Dominated Solutions ($NDS \in [0, 1]$) [4] calculates number of non-dominated solutions concerning A itself compared to \mathcal{P} , i.e., $NDS = \frac{|\{a \in A : a \in \mathcal{P}\}|}{|A|}$

The two metaheuristic algorithms considered in this paper use three operators to explore the search space. FPA uses local and global pollination operators [25], and SPEA uses recombination and mutation operators to update each individual at every iteration [26]. They use a selection operator to collect the top-scoring solutions using the definition of Pareto dominance (c.f. Def. (1)) and pass them to the next iteration. After pre-defined stopping criteria, the algorithms output the final Pareto optimal front.

Algorithm (2) presents the DSA-based multi-objective optimization performed at the compiler level. To initialize the algorithm, we recognize and collect all the *memObj* by performing standard control flow and depth-first analyses within WCC. Then, we perform the liveness analysis to determine the *live* ranges of the said *memObj*. To perform the multi-objective optimization, we need to initialize the metaheuristic algorithm. As mentioned

Algorithm 2 Multi-Objective DSA-based optimization.

```

1: Collect memObj, perform Liveness Analysis, and randomly initialize initial population of size  $N$ 
2: for  $n = 1 : N$  do
3:   Call Algorithm (1)
4: while Stopping criteria is not reached do
5:   Update Individual using respective update operators
6:   for Each updated Individual do
7:     Call Algorithm (1)
8:   Update to next generation using selection operator
9: return Pareto-optimal solution set

```

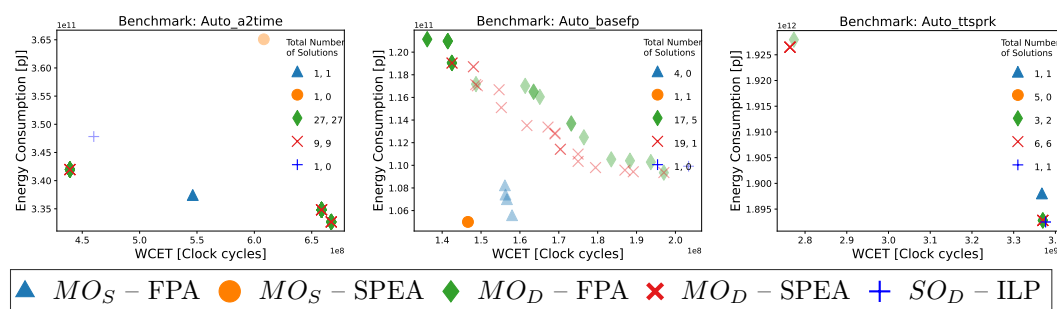
before, in this paper, we use both FPA and SPEA algorithms separately to solve the optimization problem. We randomly initialize the initial population of size N and then call the Algorithm (1) to generate dynamically allocated code and analyze the individuals. Moreover, a maximum number of generations is set as the stopping criterion, as we want the metaheuristic algorithm to terminate at some point. The metaheuristic algorithm updates the individuals at every generation and calls the Algorithm (1) to collect their objective values. Based on the objective values, the selection operator uses Pareto-dominance to select the population for the next generation. Once the stopping criterion is reached, the algorithm provides the Pareto-optimal solutions.

5 Evaluations

To the best of our knowledge, this is the first attempt to solve a compiler-level DSA-based multi-objective optimization problem that simultaneously trades multiple objectives. Therefore, we use SO_D referenced from [24] and MO_S [12] as the base for comparisons in the following evaluations. These approaches are implemented within the WCC framework, where WCET and energy analyses are performed using `aiT v21.04i` [1] and `EnergyAnalyser v21.04i` [23], respectively. WCC generates the DSA code for ARMv7-based architecture that consists of separate $ISPM$ and $DSPM$. The evaluations compare MO_D , MO_S , and SO_D in terms of the final Pareto optimal sets and the quality of obtained solutions. As SO_D is a single-objective optimization problem, we solve SO_D to minimize WCET and perform the energy analysis of the final solution to obtain both WCET and energy values.

We used benchmark suites offered by the `Embedded Microprocessor Benchmark Consortium (EEMBC)` [18] during evaluations. While performing evaluations, we use `-O0` optimization flag to turn off any other compiler-level optimization and avoid their influence on our results. We adjust the $ISPM$ and $DSPM$ sizes individually to 60% relative to the benchmark size to increase the pressure on the optimization. Moreover, we assume that the benchmarks will fit within the Flash memory.

While solving MO_D and MO_S using FPA, the switch probability is set to 0.8, as the probability of global pollination is lower than local pollination in nature. The positive integer λ for the standard gamma function and the scaling factor γ are set to 1.5 and 0.1, respectively [25, 21]. For SPEA, the size of the external population set is set to 10, which is equal to the size of each generation's population. The recombination and mutation probabilities are set to 0.8 and 0.2, respectively. The size of the population for each generation and the maximum number of generations for both algorithms are set to 10 and 80, respectively. The results obtained during these evaluations are valid for the algorithm parameters described above. For SO_D , there is no need to set any parameters.



■ **Figure 2** Solutions Obtained from MO_S , MO_D , and SO_D optimization runs.

5.1 Pareto Fronts

Fig. (2) presents the solutions found by MO_S , MO_D , and SO_D . For the sake of brevity, this subsection presents the solutions for 3 randomly chosen benchmarks.¹ The final Pareto fronts in these figures are represented using a 2D scatter plot. The x -axis and the y -axis represent WCET and energy consumption, respectively. The legend at the bottom of the figure represents the optimization runs to which the solutions belong. The legend at the top-right corner of each subfigure shows the total number of solutions returned by respective optimization runs. The same legend indicates the number of solutions on \mathcal{P} out of the total solutions. The darker-colored solutions in the figure represent the solutions on \mathcal{P} , and the fainter version of those colors represents the solutions returned by each optimization run.

For `Auto_a2time`, all solutions obtained by MO_D -FPA and MO_D -SPEA lie on \mathcal{P} . MO_S -FPA found 1 solution that is on \mathcal{P} , and the one obtained using MO_S -SPEA is not on \mathcal{P} . The 1 solution obtained by SO_D -ILP does not lie on \mathcal{P} . As SO_D is a single-objective optimization, it always outputs a single solution in the end. In the case of `Auto_basefp`, MO_D found 17 and 19 solutions using FPA and SPEA, out of which only 5 and 1 lie on \mathcal{P} , respectively. Furthermore, MO_S -SPEA found 1 solution on \mathcal{P} , and solutions obtained MO_S -FPA and SO_D -ILP do not lie on \mathcal{P} . For `Auto_ttsprk`, except for MO_S -SPEA, other 4 optimization runs found solutions on \mathcal{P} . For this benchmark, SO_D -ILP performed better than MO_S -SPEA.

For all benchmarks, we compared the total number of solutions found by respective approaches and the total number of those solutions on their final Pareto front \mathcal{P} . In that case, MO_S using FPA and SPEA had, on average, 22.92% and 32% solutions on their \mathcal{P} , and MO_D using FPA and SPEA had, on average, 51.44% and 36.75% solutions on their \mathcal{P} , respectively. Moreover, SO_D found solutions on \mathcal{P} for only 2 benchmarks. Furthermore, we calculate the contribution of each approach to the total number of solutions on \mathcal{P} for all benchmarks. In that case, MO_S -FPA and MO_S -SPEA contributed, on average, 3.62% and 5.26% solutions, MO_D -FPA and MO_D -SPEA contributed, on average, 70.4% and 20.1% solutions, and SO_D -ILP contributed, on average 0.66% solutions to the total number of solutions on \mathcal{P} for all benchmarks.

5.2 Quality Indicators

To evaluate and compare the quality of the proposed MO_D , we use three quality indicators, namely *Coverage* (C) (c.f. Def. (3)), *Non-Dominated Ratio* (NDR) (c.f. Def. (4)), and *Non-Dominated Solutions* (NDS) (c.f. Def. (5)). From these definitions, we can say the

¹ All the remaining figures can be made available at the readers' request.

following: The lower the value of \mathcal{C} , the better the approach, and the higher the values of NDR and NDS , the better the approach. Table (1) shows \mathcal{C} , NDR , and NDS indicators for all the evaluated benchmarks. For each benchmark, the table presents the values of the quality indicators for both MO_S , MO_D , and SO_D . MO_S and MO_D used FPA and SPEA algorithms, and SO_D used ILPs to solve the optimization problem. Under each quality indicator column, we have compared their values, and for each benchmark, the better quality metric is highlighted in bold in the table.

MO_S -FPA, MO_S -SPEA, and SO_D -ILP found better or indifferent solutions in terms of \mathcal{C} for 9, 7, and 2 benchmarks, respectively. MO_D -FPA and MO_D -SPEA found better or indifferent solutions for 9 and 10 benchmarks, respectively, in terms of \mathcal{C} . Therefore, we can say that, in terms of \mathcal{C} , MO_S and MO_D using FPA performed equally, MO_D -SPEA performed the best, and SO_D -ILP performed the worst. MO_S -FPA and MO_S -SPEA found either better or indifferent solutions in terms of NDR and NDS for 5 and 7, and 9 and 7 benchmarks, respectively. MO_D -FPA and MO_D -SPEA found either better or indifferent solutions in terms of NDR and NDS for 9 and 6, and 9 and 10 benchmarks, respectively. SO_D -ILP found either better or indifferent solutions for 0 and 2 benchmarks in terms of NDR and NDS , respectively. In terms of NDR , MO_D -FPA performed best, and MO_D -SPEA performed best in terms of NDS . From overall evaluations, we can say that, in general, MO_D performed slightly better than MO_S and much better than SO_D .

Finally, we also calculated the WCET and energy overheads that the MO_D solutions incur due to the dynamic allocation of memory objects during runtime using `memcpy()`. For all benchmarks, `memcpy()` functions contributed 24.39% and 22.65% to the total WCET and energy consumption, respectively. Therefore, even with a very simple and unsophisticated implementation of the `memcpy()` function that is actively executed by the processor, we obtained slightly better quality solutions using MO_D over MO_S . Furthermore, we can see that MO_D clearly outperforms SO_D . Our next steps will focus on offloading the processor from the dynamic copying of memory objects by exploiting Direct Memory Access (DMA). This way, we expect that our proposed MO_D will outright outperform MO_S .

6 Conclusion

In this paper, we proposed a novel compiler-level DSA-based multi-objective optimization that simultaneously minimizes the WCET and energy consumption of the program. The DSA model proposed in this paper handles the dynamic movement of memory objects. Moreover, we extended the WCET and energy analyses framework within the compiler to handle analyses of such dynamically allocated code. Finally, we proposed a DSA-based multi-objective optimization framework. To solve the DSA-based multi-objective optimization problem, we used two metaheuristic algorithms, namely, FPA and SPEA. We evaluated and compared the results of the proposed optimization with MO_S and SO_D using real-world benchmark suites from EEMBC. Evaluations showed MO_D provided more solutions on the final Pareto front than MO_S . Moreover, MO_D clearly outperformed SO_D . The MO_D solutions consist of `memcpy()` overheads, still, the evaluation showed that the proposed approach can provide slightly better solutions than the well-established MO_S approach.

This paper is the first step toward compiler-level DSA-based multi-objective optimization. The next step would be to improve the approach proposed in this paper by reducing the overheads. In this paper, we saw that the overheads in terms of WCET and energy due to `memcpy()` are significant. Therefore, in the future, we will explore methods to decrease

■ **Table 1** Performance Metrics for MO_S , MO_D and SO_D .

Benchmarks	Coverage						NDR						NDS					
	MO_S		MO_D		SO_D	MO_S		MO_D		SO_D	MO_S		MO_D		SO_D			
	FPA	SPEA	FPA	SPEA	ILP	FPA	SPEA	FPA	SPEA	ILP	FPA	SPEA	FPA	SPEA	ILP			
Auto_a2time	0	1	0	0	1	0.03	0	0.73	0.24	0	1	0	1	1	0			
Auto_aifft	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Auto_aifrf	1	1	0	1	1	0	0	1	0	0	0	0	1	0	0			
Auto_aifft	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Auto_basefp	1	0	0.71	0.95	1	0	0.14	0.71	0.14	0	0	1	0.29	0.05	0			
Auto_bitmnp	0	1	1	1	1	1	0	0	0	0	1	0	0	0	0			
Auto_cacheb	1	0	1	1	1	1	0	1	0	0	0	1	0	0	0			
Auto_canldr	0	1	1	1	1	1	0	0	0	0	1	0	0	0	0			
Auto_idctrn	1	1	0	0	1	0	0	0.8	0.2	0	0	0	1	1	0			
Auto_iirflt	0	0.5	1	0.5	0	0.27	0.27	0	0.36	0.1	1	0.5	0	0.5	1			
Auto_matrix	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Auto_pntrch	1	1	0	0	1	0	0	0.91	0.1	0	0	0	1	1	0			
Auto_puvmod	1	1	0	0	1	0	0	0.91	0.1	0	0	0	1	1	0			
Auto_rspeed	0	1	0	0	1	0.14	0	0.14	0.71	0	1	0	1	1	0			
Auto_tblock	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0			
Auto_ttsprk	0	1	0.33	0	0	0.1	0	0.2	0.6	0.1	1	0	0.67	1	1			
Netw_ip_pktcheck	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0			
Netw_ospfv2	1	0	1	1	1	0	1	0	0	0	0	1	0	0	0			
Netw_routelookup	1	1	1	0	1	0	0	0	1	0	0	0	0	1	0			
Tele_autocor	0	0	1	1	1	0.5	0.5	0	0	0	1	1	0	0	0			
Tele_conven	1	1	0	0	1	0	0	0.78	0.22	0	0	0	1	1	0			
Tele_fbital	0.5	0.33	0.28	0.5	1	0.04	0.08	0.54	0.33	0	0.5	0.67	0.72	0.5	0			
Tele_fft	0	1	0	1	1	0.5	0	0.5	0	0	1	0	1	0	0			
Tele_viterb	0	0.5	1	1	1	0.5	0.5	0	0	0	1	0.5	0	0	0			

the overhead incurred due to *memcpy()* and improve the MO_D solution quality even more. Currently, we are integrating DMA support within WCC to use it in conjunction with our DSA model for dynamically copying code during runtime.

References

- 1 AbsInt Angewandte Informatik, GmbH. aiT Worst-Case Execution Time Analyzers, 2021.
- 2 Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.
- 3 ARM Limited. *ARM Compiler armasm User Guide Version 5.06*, 2010-2016.
- 4 Sanghamitra Bandyopadhyay and Arpan Mukherjee. An algorithm for many-objective optimization with reduced objective computations: A study in differential evolution. *IEEE Transactions on Evolutionary Computation*, 19(3):400–413, 2014.
- 5 Jean-Francois Deverge and Isabelle Puaut. WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In *ECRTS*, pages 179–190, 2007.
- 6 Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.
- 7 Poletti Francesco, Paul Marchal, David Atienza, et al. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.
- 8 Michael R Garey, Ronald L Graham, and Jeffrey D Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 143–150, 1972.
- 9 Chi-Keong Goh and Kay Chen Tan. A competitive-cooperative coevolutionary paradigm for dynamic multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 13(1):103–127, 2008.
- 10 Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174. ACM, 2008.

- 11 Shashank Jadhav and Heiko Falk. Multi-objective optimization for the compiler of real-time systems based on flower pollination algorithm. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*, pages 45–48, 2019.
- 12 Shashank Jadhav and Heiko Falk. Approximating wcet and energy consumption for fast multi-objective memory allocation. In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, pages 162–172, 2022.
- 13 Mahmut Kandemir, J Ramanujam, Mary Jane Irwin, et al. Dynamic management of scratchpad memory space. In *DAC*, pages 690–695, 2001.
- 14 Yooseong Kim, David Broman, and Aviral Shrivastava. WCET-Aware Function-Level Dynamic Code Management on Scratchpad Memory. *ACM TECS*, 16(4):1–26, 2017.
- 15 Yu Liu and Wei Zhang. Scratchpad memory architectures and allocation algorithms for hard real-time multicore processors. *KIISE JCSE*, 9(2):51–72, 2015.
- 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(21):1437–1458, 2011. DOI 10.1002/spe.1079.
- 17 Kateryna Muts and Heiko Falk. Multi-Criteria Function Inlining for Hard Real-Time Systems. In *RTNS*, pages 56–66, 2020. DOI 10.1145/3394810.3394819.
- 18 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, 2009.
- 19 Robert Pyka, Christoph Faßbach, Manish Verma, et al. Operating System Integrated Energy Aware Scratchpad Allocation Strategies for Multiprocess Applications. In *SCOPES*, 2007.
- 20 Meikang Qiu, Zhi Chen, Zhong Ming, et al. Energy-Aware Data Allocation With Hybrid Memory for Mobile Cloud Systems. *IEEE ISJ*, 11(2), 2017.
- 21 Douglas Rodrigues, Xin-She Yang, André Nunes De Souza, and João Paulo Papa. Binary flower pollination algorithm and its application to feature selection. In *Recent advances in swarm intelligence and evolutionary computation*, pages 85–100. Springer, 2015.
- 22 Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-driven dynamic data scratchpad management with compiler-directed prefetching. In *ECRTS*, 2017.
- 23 TeamPlay Consortium. Deliverable D4.5 - Report on Energy Usage Analysis and on Prototype - Version 1.0, 2020.
- 24 Manish Verma and Peter Marwedel. Scratchpad Overlay Approaches for Main/Scratchpad Memory Hierarchy. In *Advanced Memory Optimization Techniques for Low-Power Embedded Processors*, pages 83–119. Springer, 2007.
- 25 Xin-She Yang, Mehmet Karamanoglu, and Kingshi He. Flower pollination algorithm: a novel approach for multiobjective optimization. *Engineering Optimization*, 46(9):1222–1237, 2014.
- 26 Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*, volume 63. Citeseer, 1999.