Optimizing Code Generation Efficiency Using the Polyhedral Model

SACI Abdallah*
Computer Science Department, University of BATNA 2, Algeria.
SEGHIR Rachid †
Computer Science Department, University of BATNA 2, Algeria.

Abstract

Optimizing scientific programs is critical for enhancing performance in modern computing systems, particularly in applications with stringent resource constraints such as embedded systems and parallel computing environments. In this context, the polyhedral model techniques have allowed to significantly advance the field of affine-loop-nest code generation by effectively leveraging parallelism and optimizing data locality. The present work proposes a novel approach based on the Maximal Parametric Inner-Box (MPIB) approximation algorithm, which shows promise in optimizing code generation performance. The basic idea is to introduce a new MPIB-driven transformation of the CLooG's mathematical representation of the source code, aiming at reducing the costly function calls generated during loop traversal. This leads to a significant enhancement in code performance, particularly evident with larger parameter values, where gains of up to 20% are achievable in certain cases. The preliminary results highlight notable improvements in execution time over existing techniques.

Key Words: Code Generation; Polyhedral Model; Code Optimization and Parallelization; Parametric Inner-Box; CLooG.

1 Introduction

Compiling and optimizing computer programs are of crucial importance to maximize the hardware-resource utilization and to enhance the application performances. Generating optimized code for nested loops is one of the most challenging and significant tasks in the field of code generation. This area continues to evolve, driven by the relentless pursuit of optimal performance and efficiency in modern computing systems including embedded systems that frequently operate under strict resource constraints and demand high performance for real-time applications. Additionally, the rise of parallel computing has necessitated the development of methods that can efficiently exploit parallelism inherent in loop nests.

In this context, the polyhedral model has emerged as a powerful framework, offering sophisticated tools for analyzing and optimizing affine loop nests. Through its mathematical foundations, this model provides a structured approach enabling automatic identification of parallelism, vectorization, cache locality improvement, and efficient code generation, which constitutes the main focus of the current investigation. The ability to automatically discover and leverage parallelism is particularly important in the era of multi-core processors and distributed computing environments, where parallel execution can lead to substantial performance gains.

Polyhedral code generation techniques, including CADGen (Continuous Automatic Differentiation Code Generator), CodeGen (Code Generator), CodeGen+ (Enhanced Code Generator), isl (Integer Set Library), and ClooG (Chunky Loop Generator)[2, 6, 5, 22, 21], have revolutionized the field of code generation for complex nested loops. These techniques enable considerable improvements in the performance of the generated code by effectively harnessing parallelism and optimizing data locality.

However, these methods often face limitations due to the computational overhead introduced by complex function calls in loop bounds, such as *min, max, ceil*, and *floor* functions. These operations, while essential for accurate bounds calculation, can become a bottleneck, particularly in performance-sensitive environments such as embedded systems and real-time applications.

In this article, we introduce a new approach for generating efficient code based on the concept of Maximal Parametric Inner-Box (MPIB). The main research question that our work addresses is: how can code generation be optimized to reduce computational load while maintaining accuracy? The key questions we seek to answer are: (i) What parametric factors significantly influence code generation efficiency? (ii) How can maximal inner-box approximation be applied to achieve this optimization?

Our method involves identifying and characterizing the parametric maximal inner box for each polyhedral set. This approach is used to explore and reorganize the loop transformation and optimization space, ensuring efficient utilization of computational resources. Consequently, we manipulate regular iteration domains to minimize expensive function calls during loop traversal whenever possible, which

*LaSTIC, Laboratory, Computer Science Department, University of BATNA 2, Algeria. Email: a.saci@univ-batna2.dz.
†LaSTIC, Laboratory, Computer Science Department, University of BATNA 2, Algeria. Email: r.seghir@univ-batna2.dz.

ultimately results in an improvement in the overall performance of the generated code. This improvement directly translates to better utilization of the limited computational resources.

The preliminary results of this work reveal that the proposed method offers significant advantages in terms of execution time compared to CLooG 0.18.4. This finding paves the way for further investigation of the impact of this new approach in the domain of nested loop optimization. Note that the MPIB approach can also be utilized in other real-world applications which are out of the focus of the present work, such as the reachability of hybrid dynamical systems, where it is crucial to know if the system can reach critical regions or stay within safe sets[19].

The remainder of this article is organized as follows: Section 2 delves into the foundational concepts of code generation using the polyhedral model, including its representation of programs and techniques for optimization. In Section 3, we present our approach, elucidating the mathematical foundations of the Maximal Parametric Inner Box (MPIB). The practical application of the MPIB approximation approach in generating efficient code is discussed in Section 4. Section 5 provides an illustrating example of our method. In section 6, we compare our approach with prior work. Finally, Section 7 presents a conclusion, summarizing our findings and highlighting avenues for future research.

2 BACKGROUND

In this section, we introduce the fundamental principles of code generation in the context of the polyhedral model. Our discussion delves into how programs are represented within this framework, emphasizing the role of CLooG tool (Chunky Loop Generator) in efficiently generating code from polyhedral representations.

2.1 Polyhedral Model

The polyhedral model is a powerful mathematical and geometrical framework for the analysis and optimization of programs through linear algebra and polyhedral geometry [5, 8]. It includes loop transformations, data restructuring, and various other techniques aimed at enhancing program performance [9, 13, 25, 24, 23, 4, 7]. This form of optimization techniques usually targets improving data locality and parallelism in code, which can greatly impact the overall efficiency of a program. Over the years, the polyhedral model has been proven successful in a wide range of cases and has become a fundamental tool in program optimization.

The polyhedral model proceeds through three primary steps. Initially, it expresses the original code into a geometric representation, associating each statement with a set of polyhedra. Next, it performs geometric transformations within this representation. Finally, it translates the set of polyhedra back into generated code. In this article, we focus on this later step where we target generating an efficient code based on our maximal parametric inner-box approach presented in section 3.

2.1.1 Polyhedral representation of programs

In the polyhedral model, a program is represented by an iteration domain, which is a set of affine functions mapping each statement (or point) in the original code to a point in the iteration domain [1, 16, 18].

Kuck [11] showed that the iteration domain of a loop nest (a set of nested loops), with affine lower and upper bounds, can be described by a polyhedron bounded by a set of half-spaces. Each half-space corresponds to a lower or upper bound on an index. The dimension of the polyhedron thus defined is equal to the depth of the loop nest (the number of its indices). Finally, each point with integer coordinates (integer vector) inside the polyhedron corresponds to an iteration of the loop nest. When the number of iterations is not fixed (cannot be determined at compile time), we refer to parametric loop nests. These are loop nests that contain symbolic constants (parameters) in the affine expressions of their bounds. A loop nest where all instructions are at the innermost level is called perfect. The general form of a perfect loop nest of depth d is:

for i_1=$l_1(p)$ to $u_1(p)$
 for i_2=$l_2(i_1,p)$ to $u_1(i_1,p)$
  ....
   for i_d=$l_d(i_1,i_2, \dots\ i_{d-1,p})$ to $u_d(i_1,i_2, \dots\ i_{d-1},p)$
    ....

where i_j (j = 1,..., d) are the indices of the loop nest, p is a parameter vector, and l_j, u_j (j = 1,..., d) are affine functions. When the loop nest is not perfect, instructions can appear at any depth level.

2.1.2 Example

Consider the following piece of code (loop nest):

for(i=*1*; $i \leq n$; i++)
 for(j=*1*; $j \leq i+m$; j++)
  S(i,j);

The iterations of this loop nest correspond to the integer-coordinate points of the parametric polytope $P(p)$ as follows:

$$P(p) = \left\{ \binom{i}{j} \in \mathbf{Q}^2 \mid 1 \leq i \leq n \ \wedge\ 1 \leq j \leq i+m \right\}$$

where $p = [n\ m]$ is an integer parameter vector. The graphical representation of the iterations of this loop nest, when $p = [n\ m]$=[5 2], is shown in Figure 1.

2.2 Code generation using the polyhedral model

Code generation has seen significant development thanks to the algorithm introduced by Quilleré et al.[15]. Since then, many research efforts have been conducted to enhance the quality of the generated code [2, 10, 14, 17, 20].

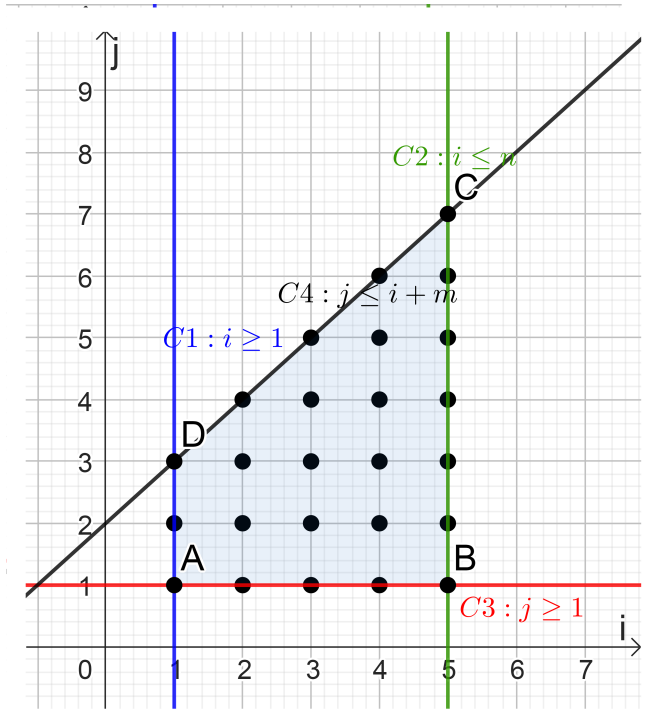

Figure 1: Representation of loop nest iterations for example 1 (with n=5 and m=2).

Quilleré's algorithm involves calculating a disjoint union of polyhedra at each recursion level across dimensions, and then generating code for each resulting subset sequentially.

Although this approach results in more extensive output code, it lowers execution complexity. This reduction is crucial for minimizing energy consumption and optimizing performance in applications with strict resource limitations, like embedded systems. However, some tests and multiple loop bounds requiring calls to *ceil/floor* and *min/max* functions are not eliminated.

This algorithm is implemented in the widely used code generation tool CLooG (Chunky Loop Generator) [2], which incorporates various enhancements aimed at preventing large code generation. These improvements include reducing the complexity of splitting, the number of scanned subsets, and the size of the generated code, all while maintaining performance [17].

2.3 CLooG: Chunky Loop Generator

CLooG (Chunky Loop Generator) is a crucial tool in the field of polyhedral code generation. It enables the efficient translation of polyhedral representations into optimized nested loop structures, which is essential for maximizing performance in resource-constrained environments or those requiring high levels of parallelism.

In the following, we give an overview of the primary algorithm used in CLooG, as initially proposed by Cedric Bastoul [2].

The CLooG tool takes as input a union of polyhedra representing the source program. Each statement of the program is thus represented by a subset of polyhedra and a set of scheduling functions. Applying these functions to the integer points of the associated polyhedra results in a new list of polyhedra that the resulting code must scan.

According to the technique proposed by LeVerge [12], the set of integer points in a polyhedron is represented as a ZPolyhedron[1].

In order to generate a loop code, the CLooG algorithm starts by computing the projection of polyhedra at dimension ($d = 1$), subsequently separating them into an ordered list of disjoint polyhedra. It then scans this list to produce code for the outermost loops (level-one loops). These disjoint polyhedra are then projected onto the second dimension ($d = 2$) to generate code for level-two loops. CLooG iterates recursively across the remaining dimensions (levels 3, 4, ...) to generate loop codes at the corresponding levels. The detailed algorithm is given in Algorithm 1.

In step 5, the algorithm computes the lower bound and the stride for each loop level ($d \in$ 1,2, ..., n) defined by its subdomains (polyhedra). Then, it merges inner polyhedra whenever possible in step 5(b.i) in order to reduce the code size. In step 5(b.ii), the function is recursively called for the next dimension ($d+1$) by intersecting the context domain with the bounds of the currently generated loop. Step 7 involves reuniting certain point polyhedra with their host polyhedra, from which they were separated in the previous step, with the aim of minimizing the overall size of the generated code [17].

Although CLooG excels at managing polyhedral sets to produce high-performance code, it has limitations, particularly with the generated *min, max, ceil,* and *floor* function calls in loop bounds, which can become computationally expensive. In the following sections (3 and 4), we will show that our MPIB-based method reduces these costs by simplifying loop bounds, thereby reducing function calls and enhancing the overall execution time of the generated code.

3 Maximal Parametric Inner-Box (MPIB) approximation approach

In this section, we propose a new approach for approximating the maximal parametric inner box based on the method of Bemporad et al. [3]. In their work, and starting from a non-parametric polytope P, the authors search for two collections of boxes (I and E) such that:

- The interiors of the boxes in each collection do not overlap,
- The union of all boxes in I is contained in P.
- The union of all boxes in E contains P.

Note that in the current work, we are interested in determining only one approximate maximal inner box within a

[1] A ZPolyhedron is the intersection of an integral lattice and a polyhedron.

Algorithm 1: CLooG's code generation algorithm [2].

Data: A polyhedron list $(TS_1, ..., TS_n)$, a context C, the current dimension d.

Result: Code scanning the polyhedra inside the input list.

Begin

1. Intersect each polyhedron $P_i \in T_{Si}$ with the context C.
2. Compute the projection P_i onto the outermost d dimensions for each resulting polyhedron T_{S_i}, and consider the new list where T_{S_i} is replaced by P_i.
3. Separate the list of resulting projections P_i from step 2 into a new list of non-overlapping polyhedra.
4. Order each list of non-overlapping polyhedra representing the projection P_i, from step 3, in the lexicographical order.
5. For each polyhedron $P \rightarrow (T_{Sp}, \cdots, T_{Sq})$ in the list:
   (a). Compute the stride and the lower bound by looking for stride constraints in the $(T_{Sp}, \cdots, T_{Sq})$ list.
   (b). While there is a polyhedron in $(T_{Sp}, \cdots, T_{Sq})$:
      (i). Merge adjacent polyhedra scanning the same statements in a new list.
      (ii). Recurse for the new list with the new loop context $C \cap P$ and the next dimension $d+1$.
6. Apply steps 2 to 4 of the algorithm to the inside list in order to eliminate dead code, for each polyhedron P inside the list.
7. Reduce code size by making all possible unions of host polyhedra with point polyhedral.
8. Return the code scanning the polyhedron list.

End.

2-dimensional parametric polytope $P(p)$ that has one parameter $(p = [n])^2$. This box will be used in section 4 to generate an efficient code based on CLooG Algorithm. Our method can be succinctly described as follows:

Let $P(n)$ be the parametric polytope defined by:

$$P(n) = \{X \in \mathbf{R}^2 : AX \leq Bn + b\}$$

And let :

- A^+ : be the positive matrix of A.
- A_1^+ : be the first column of A^+.
- A_2^+ : be the second column of A^+.

To determine an approximation of the maximal parametric box included in $P(n)$, we start by assigning distinct values to n in order to obtain different instances of the polytope $P(n)$. For each instance of $P(n)$, we determine the maximal inner box based on the method proposed by Bemporad et al. [3], which involves the following steps:

[2]Our method can be extended to address problems involving higher dimensions and additional parameters.

1. Solve LP1 to find r_1, the maximum ratio along the first dimension.
   $LP1 : r_1 = max\{r : AX + A_1^+ r \leq Bn + b\}$.
2. Solve LP2 to find r_2, the maximum ratio along the second dimension.
   $LP2 : r_2 = max\{r : AX + A_2^+ r \leq Bn + b\}$.
3. Solve LP3 to determine the scaling factor λ^* which maximizes the box dimensions while ensuring it stays within $P(n)$.
   $LP3 : \lambda^* = max\{\lambda : AX + A^+ r\lambda \leq Bn + b\}$, with $r = [r_1 r_2]$.

The solution of $LP3$ is defined by: (X^*, λ^*), with $X^* = [i^* j^*]^t$.

For a given instance of the polytope $P(n)$, i.e for a given value of n, the MPIB is defined by its two extremal non-parametric points (V_1 and V_2) such that:

- $V_1(i_{V_1}, j_{V_1})$, with : $i_{V_1} = i^*$ and $j_{V_1} = j^*$.
- $V_2(i_{V_2}, j_{V_2})$, with: $i_{V_2} = i^* + r_1.\lambda^*$ and $j_{V_2} = j^* + r_2.\lambda^*$.

These points mark the endpoints of the approximate largest box included in the considered instance of polytope $P(n)$. To compute the parametric coordinates of the extremal points of the MPIB included in the parametric polytope $P(n)$, we need to compute a regression line for each of the four coordinates. These lines are given by the following equations:

$$\begin{aligned} i_{V_1}(n) &= \alpha_1.n + \beta_1 \\ j_{V_1}(n) &= \alpha_2.n + \beta_2 \\ i_{V_2}(n) &= \alpha_3.n + \beta_3 \\ j_{V_2}(n) &= \alpha_4.n + \beta_4 \end{aligned}$$

The parametric coordinates of $V_{1(n)}(i_{V_{1(n)}}, j_{V_{1(n)}})$ and $V_{2(n)}(i_{V_{2(n)}}, j_{V_{2(n)}})$ define the parametric maximal inner box of $P(n)$. Indeed, in order to determine this box, it suffices to find its two extremal points V_1 and V_2 having the lowest, respectively highest coordinates as shown in Figure 5. Our approach of approximating the MPIB is described in Algorithm 2.

Example:

Let $P(p)$ be the parametric polytope defined by the following inequations:

$$P(p) = \begin{cases} -i + j & \leq -2 \\ i + j & \leq n \\ 2i - 6j & \leq n + 4 \\ -9i + 18j & \leq n - 2 \\ n & \geq 20 \end{cases}$$

$P(p)$ can be rewritten as follows:

$$P(p) = \{X \in \mathbf{R}^2 : AX \leq Bp + b\}, \ where :$$

$$X = \begin{bmatrix} i \\ j \end{bmatrix}, A = \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ 2 & -6 \\ -9 & 18 \end{bmatrix}, b = \begin{bmatrix} -2 \\ 0 \\ 4 \\ -2 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}, and \ p = \begin{bmatrix} n \end{bmatrix},$$

A^+,A_1^+, and A_2^+, are given as follows:

$$A^+ = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 0 \\ 0 & 18 \end{bmatrix}, A_1^+ = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 0 \end{bmatrix}, and\ A_2^+ = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 18 \end{bmatrix}$$

By assigning different values to the parameter n (instantiation of $P(n)$) and solving the linear programs $LP1$, $LP2$, and $LP3$ below, we obtain the coordinates of the two extremal points V_1 and V_2 of the approximate MPIB of $P(n)$, as shown in Table 1.

$$LP1 : r_1 = max \left\{ r : \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ 2 & -6 \\ -9 & 18 \end{bmatrix} X + \begin{bmatrix} 0 \\ 1 \\ 2 \\ 0 \end{bmatrix} r \leq \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} n + \begin{bmatrix} -2 \\ 0 \\ 4 \\ -2 \end{bmatrix} \right\}$$

$$LP2 : r_2 = max \left\{ r : \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ 2 & -6 \\ -9 & 18 \end{bmatrix} X + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 18 \end{bmatrix} r \leq \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} n + \begin{bmatrix} -2 \\ 0 \\ 4 \\ -2 \end{bmatrix} \right\}$$

$$LP3 : \lambda^* = max \left\{ \lambda : \begin{bmatrix} -1 & 1 \\ 1 & 1 \\ 2 & -6 \\ -9 & 18 \end{bmatrix} X + \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 2 & 0 \\ 0 & 18 \end{bmatrix} r\lambda \leq \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} n + \begin{bmatrix} -2 \\ 0 \\ 4 \\ -2 \end{bmatrix} \right\}$$

, where $r = [r_1\ r_2]$

Then, we use Microsoft Excel solver to determine the four regression lines defining the four parametric coordinates of points $V_{1(n)}(i_{V_1}(n), j_{V_1}(n))$ and $V_{2(n)}(i_{V_2}(n), j_{V_2}(n))$ from the instances of $V_1(i_{V_1}, j_{V_1})$ and $V_2(i_{V_2}, j_{V_2})$:

$$i_{V_1}(n) = 0,384258945560599.n - 0,318353165232634$$
$$j_{V_1}(n) = 0,084104990554686.n - 0,576570768496135$$
$$i_{V_2}(n) = 0,752314971664028.n + 0,270287695130927$$
$$j_{V_2}(n) = 0,247685028335843.n - 0,270287693456269$$

Finally, the parametric inner box is defined by the two extremal points $V_{1(n)}(i_{V_1}(n), j_{V_1}(n))$ and $V_{2(n)}(i_{V_2}(n), j_{V_2}(n))$ as illustrated in Figure 5 (for n=50).

In determining the regression lines for the coordinates of the maximal parametric inner box (MPIB), we experimented with various levels of decimal precision to ensure that each calculated point remains a valid integer point within the polytope, across a wide range of values for the parameter n. Specifically, we evaluated precision levels of 6, 8, 10, 12, and 15 decimal places in the regression expressions. The results showed that a precision of at least 15 decimal places was necessary to maintain the validity of all coordinates as integer points inside the polytope for n values ranging from the initial value up to 1,000,000. With fewer than 15 decimal places, certain coordinates occasionally fell outside the bounds of the polytope, which would compromise the accuracy of the MPIB approximation. Thus, we opted for 15 decimal places in the regression expressions to ensure that the MPIB coordinates reliably represent integer points within the polytope over the full range of parameter values considered in our study.

It is worth noting that it is possible to consider coordinates

Algorithm 2: Approximation of the Maximal Parametric Inner Box.

Data: A parametric polytope $P(n)$.
Result: MPIB
Begin
Let :
A^+ be the positive matrix of A.
A_1^+ and A_2^+ be the first and second columns of A^+, respectively.
Let :
$A_{i_{V_1}}[\,]$, $A_{j_{V_1}}[\,]$, $A_{i_{V_2}}[\,]$ and $A_{j_{V_2}}[\,]$ be the coordinate arrays of points V_1 and V_2 (for different values of the parameter).
Step 1:
$counter \leftarrow 1$;
$n \leftarrow intilal\_value$;
while $(n \leq final\_value)$ **do**
  Solve $LP1 : r_1 = max\{r : AX + A_1^+ r \leq Bn + b\}$.
  Solve $LP2 : r_2 = max\{r : AX + A_2^+ r \leq Bn + b\}$.
  Solve $LP3 : \lambda^* = max\{\lambda : AX + A^+ r\lambda \leq Bn + b\}$.
  with $r = [r_1 r_2]$.
  //The solution of $LP3$ is: (X^*, λ^*),
  //where: $X^* = [i^*\ j^*]^t$.
  // The inner box is defined by the two points
  // $V_1(i_{V_1}, j_{V_1})$ and $V_2(i_{V_2}, j_{V_2})$, where
  //$i_{V_1} = i^*, j_{V_1} = j^*, i_{V_2} = i^* + r_1\lambda^*$ and
  //$j_{V_2} = j^* + r_2\lambda^*$.
  $A_{i_{V_1}}[counter] \leftarrow i^*$;
  $A_{j_{V_1}}[counter] \leftarrow j^*$;
  $A_{i_{V_2}}[counter] \leftarrow i^* + r_1\lambda^*$;
  $A_{j_{V_2}}[counter] \leftarrow j^* + r_2\lambda^*$;
  $counter \leftarrow counter + 1$;
  $n \leftarrow n + step$; //step = 100000
end
Step 2
Determination of the four regression lines corresponding to the values stored in arrays:
$A_{i_{V_1}}[\,]$,$A_{j_{V_1}}[\,]$,$A_{i_{V_2}}[\,]$, *and* $A_{j_{V_2}}[\,]$ as follows:
  $i_{V_1}(n) \leftarrow \alpha_1.n + \beta_1$;
  $j_{V_1}(n) \leftarrow \alpha_2.n + \beta_2$;
  $i_{V_2}(n) \leftarrow \alpha_3.n + \beta_3$;
  $j_{V_2}(n) \leftarrow \alpha_4.n + \beta_4$;
The approximate MPIB is defined by the two parametric points :
  $V_{1(n)}(i_{V_1}(n), j_{V_1}(n))$ and
  $V_{2(n)}(i_{V_2}(n), j_{V_2}(n))$.
end.

Table 1: Coordinates of V_1 and V_2 for Example 1.

n	20	100	1000	10000	100000	150000	200000	1000000
iV_1	8	39	384	3843	38426	57639	76852	384259
jV_1	2	8	84	841	8410	12616	16821	84105
iV_2	16	76	753	7524	75232	112848	150464	752316
jV_2	4	24	247	2476	24768	37152	49536	247684

with fewer decimal places. However, an additional verification step is required to ensure that all coordinates lie within the polytope. If any coordinate does not satisfy this inclusion constraint, it is necessary to adjust the point by selecting the nearest valid coordinates within the polytope.

4 ENHANCING CODE GENERATION PERFORMANCE USING THE MPIB APPROXIMATION ALGORITHM

Methodology

Our methodology is based on the Maximal Parametric Inner-Box (MPIB) approximation algorithm, which aims to optimize code generation by reducing costly function calls in loop bounds. The approach involves three main steps:

1. Start by running the first three setps of CLooG algorithm to generate a polyhedral representation of the source code to be optimized.
2. Apply the MPIB approach to convert the polyhedral representation into a new form that enhances code performance.
3. Resume the CLooG algorithm from Step 4 to generate a new code using this later polyhedral representation.

In the following, we will show how the Maximal Parametric Inner Box (MPIB) approximation approach can be applied in generating effective code. The main objective of our work is to generate an efficient code using the polyhedral model. This code will be generated by combining CLooG algorithm with our method of approximating the MPIB presented in the previous section. This approach consists in modifying CLooG algorithm (Algorithm 1) immediately after step 3. In this new algorithm, we start by calling CLooG until step 3. Then we compute the approximate MPIB for each sub-polytope P_i obtained at step 3 and replace it with the following 5 sub-polytopes $P_{i_1}, P_{i_2}, P_{i_3}, P_{i_4}$, and P_{i_5}:

- $P_{i_1} = P_i \cup \{i < i_{V_1}(n)\}$,
- $P_{i_2} = P_i \cup \{i \geq i_{V_1}(n), i \leq i_{V_2}(n), j < j_{V_1}(n)\}$,
- P_{i_3}=$\{i \geq i_{V_1}(n), i \leq i_{V_2}(n), j \geq j_{V_1}(n), j \leq j_{V_2}(n)\}$ // the MPIB,
- $P_{i_4} = P_i \cup \{i \geq i_{V_1}(n), i \leq i_{V_2}(n), j > j_{V_2}(n)\}$,
- $P_{i_5} = P_i \cup \{i \geq i_{V_2}(n)\}$.

After the step of generating this new polyhedral representation of the code to be optimized, we resume CLooG algorithm from step 4. This means that, instead of generating the code for the polyhedral set given by CLooG, we do it for the new polyhedral representation based on the MPIB approximation approach. This approach offers the advantage of efficiently handling regular polyhedral sets, requiring only a few calls to *min/max* and *floor/ceil* functions. This optimization significantly improves the execution time of the generated code.

We note that the core factor in our work is the execution time, as this is critical in evaluating the efficiency of the generated code within the polyhedral model. Algorithm 3 summarizes our MPIB-based code generation method.

It should be noted that the method for determining the MPIB can, if necessary, be applied recursively to one or all of the sub-polytopes $P_{i_1}, P_{i_2}, P_{i_4}$ or P_{i_5} when the polytope is sufficiently large. Furthermore, this method can be generalized to higher dimensions, which involves solving $(d+1)$ linear programs for a dimension d.

5 ILLUSTRATING EXAMPLE

Consider the following parametric polytope:

$$P(p) = \begin{cases} -i+j & \leq -2 \\ i+j & \leq n \\ 2i-6j & \leq n+4 \\ -9i+18j & \leq n-2 \\ n & \geq 20 \end{cases}$$

The corresponding code generated by CLooG-0.18.4 for this polytope and its graphical representation are shown in Figures 2 and 3, respectively. Note that the presence of calls to the *min/max* and *floor/ceil* functions in the inner-loop bounds of the generated code results in a substantial control overhead, affecting execution time. The idea of our approach is to avoid, as much as possible, costly function calls inside loop bounds using the modified GLooG algorithm (Algorithm 3) based on the MPIB approximation method (Algorithm 2). The resulting optimized code and its corresponding iteration domain are shown in Figures 2 and 3, respectively.

6 COMPARISON WITH PRIOR WORK AND ANALYSIS

In this section, we provide a comparative analysis between our method and existing techniques, particularly focusing on the latest version of the CLooG tool (0.18.4), which has been

```
for (i=2; i≤ floord(7*n+4,8); i++) {
  for(j=max(0,ceild(2*i-n-4,6));j≤min(min(floord(9*i+n-2,18),-i+n),i-2); j++){
    S(i,j);
  }
}
```

Figure 2: Generated code by CLooG 0.18.4

Algorithm 3: Code generation using MPIB approximation approach (**Modified CLooG's algorithm**).

Data: a parametric polytope P(n).
Result: Generated Code
Begin
Step 1.
Execution of Steps 1, 2 and 3 of the ClooG algorithm (Algorithm 1) to generate a disjoint union of polytopes $S = \bigcup_{i=1}^{|S|} P_i$ corresponding to the union of input polytopes (Polyhedral representation of the source code to be optimized)
Step 2
Decomposition of each polytope $P_i \in S$ into a disjoint union of 5 sub-polytopes $P_{i_1}, P_{i_2}, P_{i_3}, P_{i_4}$, and P_{i_5}, where :

For all $l,k \in \{1,2,3,4,5\}$ and $l \neq k$: $\begin{cases} P_i = \cup_{j=1}^{5} P_{i_j} \\ P_{i_l} \cap P_{i_k} = \emptyset \end{cases}$

with:

- $P_{i_1} = P_i \cup \{i < i_{V_1}(n)\}$,
- $P_{i_2} = P_i \cup \{i \geq i_{V_1}(n), i \leq i_{V_2}(n), j < j_{V_1}(n)\}$,
- $P_{i_3} = \{i \geq i_{V_1}(n), i \leq i_{V_2}(n), j \geq j_{V_1}(n), j \leq j_{V_2}(n)\}$,
- $P_{i_4} = P_i \cup \{i \geq i_{V_1}(n), i \leq i_{V_2}(n), j > j_{V_2}(n)\}$,
- $P_{i_5} = P_i \cup \{i \geq i_{V_2}(n)\}$.

Step 3
Resume the CLooG algorithm from Step 4 to generate the code corresponding to the disjoint union of polytopes generated in the previous step (Step 2). **end.**

widely used for polyhedral code generation. While CLooG is recognized for its efficiency in generating code from polyhedral representations, our method introduces the Maximal Parametric Inner-Box (MPIB) approximation, which offers notable improvements in execution time.

One of the key differences between our approach and previous work lies in the way the loop bounds are handled. Traditional methods, including CLooG, rely heavily on the use of *min/max* and *ceil/floor* function calls, which can add significant overhead in execution. In contrast, our approach minimizes these function calls by approximating the maximal inner-box, leading to reduced computational load and enhanced performance. This difference is more significant for large

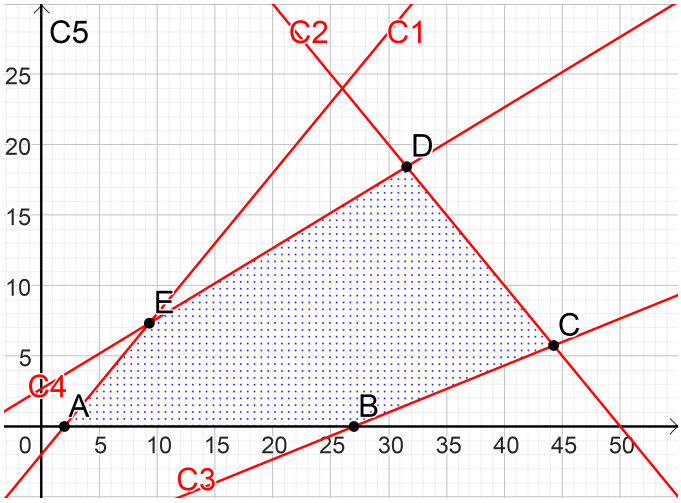

Figure 3: Graphical representation of example 1

parameter values.

In order to demonstrate the performance of our method, we consider the codes from Figures 2 and 4 generated by the original CLooG algorithm and our MPIB-based method respectively. These codes were compiled with gcc 5.4 and executed on an Intel i5 processor at 2.30GHz, with the values of the parameter n ranging from 100000 to 10000000 and a step of 100000.

Figure 6 and Table 2 present the execution times for both the standard CLooG algorithm and the proposed MPIB-based approach across varying parameter values. This comparison allows us to observe the substantial improvement in execution time when employing our approach, particularly noticeable with larger values of the parameter n. For instance, with n = 500000, the runtime for the code produced by CLooG-18.0.4 is 173.99 s, whereas our method yields a runtime of 140.87 s, resulting in a gain rate of 19.04%. This improvement is due to the decreased number of calls to *min/max* and *floor/ceil* functions in the code generated by our approach.

7 CONCLUSION AND FUTURE WORK

Optimizing code generation for nested loops is crucial in maximizing hardware resource utilization and enhancing application performance. The polyhedral model, with its sophisticated tools, is extensively employed in code generation algorithms.

The efficiency of the generated code is significantly impacted

```
for (i=2;i≤floord(9606473639*n-7958829131,25000000000);i++) {
  for (j=0;j≤min(floord(9*i+n-2,18),i-2);j++){
    S1(i,j);
  }
}
for(i=ceild(9606473639*n-7958828880,25000000000); i≤floord(47019685729*n+16892980945,
      62500000000);i++){
  for(j=max(0,ceild(2*i-n-4,6));j≤floord(42052495277*n-288285384248, 500000000000);j++) {
    S1(i,j);
  }
  for(j=max(ceild(2*i-n-4,6),ceild(42052495277*n-288285379248,500000000000));j≤
      floord(49537005667*n-54057538692,200000000000);j++){
    S1(i,j);
  }
  for(j=ceild(49537005667*n-54057536691,200000000000);j≤min(floord(9*i+n-2,18),-i+n);j++){
    S1(i,j);
  }
}
for(i=ceild(47019685729*n+16892981571,62500000000);i≤floord(7*n+4,8);i++) {
  for (j=ceild(2*i-n-4,6);j≤-i+n;j++) {
    S1(i,j);
  }
}
```

Figure 4: Generated code by our approach

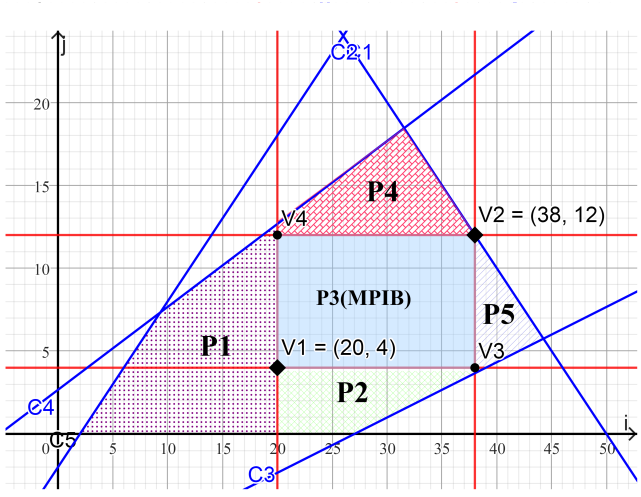

Figure 5: Separation of the polytope into 5 sub-polytopes

Table 2: Execution times for CLooG's algorithm and Our Algorithm

n	100000	200000	300000	400000	500000	600000	700000	800000	900000	1000000
CLooG(s)	6.96	27.84	62.64	111.35	173.99	250.52	340.98	445.36	564.04	689.92
MPIB(s)	5.36	21.29	50.72	90.16	140.87	202.86	276.11	360.60	456.37	563.63
Gain	1.60	6.55	11.92	21.19	33.12	47.66	64.87	84.76	107.67	126.29
Ratio(%)	22.97	23.52	19.03	19.03	19.04	19.03	19.02	19.03	19.09	18.30

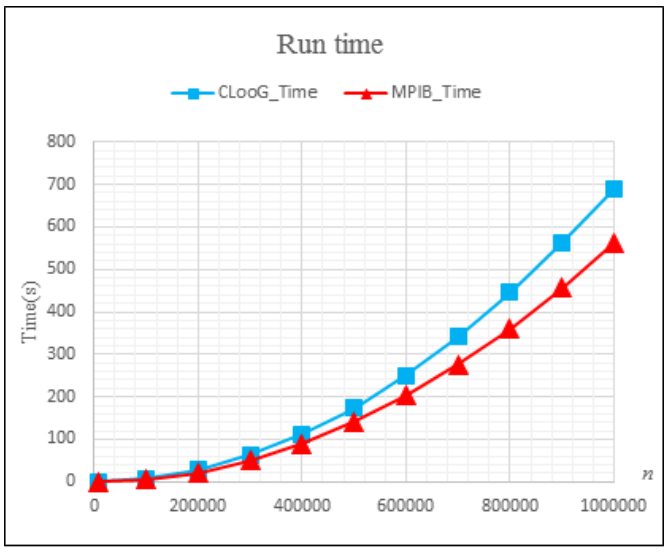

Figure 6: Execution times

by how polyhedral operations are applied to the mathematical representation of the original code.

This article presents a new approach based on the proposed parametric maximal inner-box approximation algorithm, representing a promising avenue for further enhancing code generation efficiency.

By identifying this box for each polyhedral set and performing the underling transformations, we mitigate costly function calls during loop traversal, ultimately leading a substantial improvement of the code performance, particularly with larger values of the parameter where the gain could achieve about 20% in some cases. The experimental results demonstrate notable advantages compared to existing techniques, encouraging further exploration of the potential impact of this approach in nested loop optimization.

In line with the problem statement and research contribution, our study has successfully addressed the challenge of optimizing loop-based code generation by leveraging the polyhedral model. Focusing on reducing computational overhead, particularly through the new concept of MPIB, we have provided a solution that improves execution efficiency in a way that was not previously explored.

Despite the significant improvements in execution time and efficiency achieved by the proposed MPIB-based approach, some limitations can be observed. While the method enhances performance in many cases, its effectiveness is highly dependent on the structure of the polyhedral sets being processed. In particular, when iteration domains are highly irregular or contain non-affine constraints, the approximation may not deliver optimal results. Additionally, extending the approach to higher-dimensional polyhedral sets presents challenges, as the complexity of solving additional linear programs increases significantly.

This study opens up several promising directions for future research in nested loop optimization. First, extending the proposed MPIB approach to more complex cases, such as higher-dimensional polyhedral sets dealing with deeper levels of nesting. Additionally, hybrid optimization strategies, like combining MPIB with techniques such as parametric tiling or dynamic loop transformations, could yield further performance improvements. Another key area for exploration is the integration of this method into modern compilation frameworks to facilitate its adoption by software developers and increase its usability in practical applications. Addressing these aspects would allow the proposed approach to be refined and extended to a broader range of applications in polyhedral optimization.

References

[1] A. Acharya, U. Bondhugula, and A. Cohen. “An Approach for Finding Permutations Quickly: Fusion and Dimension matching”. _eprint: abs/1803.10726. 2018.

[2] Cédric Bastoul. “Code Generation in the Polyhedral Model Is Easier Than You Think”. In: *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. Jan. 2004, pp. 7–16. ISBN: 0-7695-2229-7. DOI: 10.1109/PACT.2004.1342537.

[3] A. Bemporad, C. Filippi, and F. D. Torrisi. “Inner and outer approximations of polytopes using boxes”. In: *Comput. Geom.* 27 (2004), pp. 151–178.

[4] Wlodzimierz Bielecki, Marek Palkowski, and Maciej Poliwoda. “Automatic code optimization for computing the McCaskill partition functions”. In: Sept. 2022, pp. 475–478. DOI: 10.15439/2022F4. URL: https://annals-csis.org/Volume_30/drp/4.html (visited on 09/20/2024).

[5] U. Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*. 2008.

[6] Chun Chen. “Polyhedra scanning revisited”. en. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing China: ACM, June 2012, pp. 499–508. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254123. URL: https://dl.acm.org/doi/10.1145/2254064.2254123 (visited on 07/10/2024).

[7] Gianpietro Consolaro et al. “PolyTOPS: Reconfigurable and Flexible Polyhedral Scheduler”. In: *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Edinburgh, United Kingdom: IEEE, Mar. 2024, pp. 28–40. ISBN: 9798350395099. DOI: 10.1109/CGO57630.2024.10444791. URL: https://ieeexplore.ieee.org/document/10444791/ (visited on 09/20/2024).

[8] P. Feautrier and C. Lengauer. *Polyhedron Model*. Encyclopedia of Parallel Computing, 2011.

[9] L. Gonnord et al. “A Survey on Parallelism and Determinism”. In: *ACM Computing Surveys* 55 (2022), pp. 1–28.

[10] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. “Polyhedral AST Generation Is More Than Scanning Polyhedra”. en. In: *ACM Trans. Program. Lang. Syst.* 37.4 (Aug. 2015), pp. 1–50. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/2743016. URL: https://dl.acm.org/doi/10.1145/2743016 (visited on 07/10/2024).

[11] D. J. Kuck. *The Structure of Computers and Computations*. Vol. 1. Wiley, 1978.

[12] H. Le Verge. “Recurrences on lattice polyhedra and their applications”. 1995.

[13] L. Narmour, T. Yuki, and S. V. Rajopadhye. “Maximal Simplification of Polyhedral Reductions”. _eprint: abs/2309.11826. 2023.

[14] L. Pouchet et al. “Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time”. In: *International Symposium on Code Generation and Optimization (CGO’07)*. 2007, pp. 144–156.

[15] F. Quilleré, S. V. Rajopadhye, and D. Wilde. “Generation of Efficient Nested Loops from Polyhedra”. In: *International Journal of Parallel Programming* 28 (2000), pp. 469–498.

[16] W. Ranasinghe et al. “PCOT: Cache Oblivious Tiling of Polyhedral Programs”. _eprint: abs/1802.00166. 2018.

[17] H. Razanajato, V. Loechner, and C. Bastoul. “Splitting polyhedra to generate more efficient code”. In: *IMPACT 2017, 7th International Workshop on Polyhedral Compilation Techniques*. Jan. 2017.

[18] L. V. Thekkekara and B. Cai. “Significant efficiency enhancement in thin film solar cells using laser beam-induced graphene transparent conductive electrodes”. _eprint: Applied. 2018.

[19] F. D. Torrisi and A. Bemporad. “Discrete-time hybrid modeling and verification”. In: *Proceedings of the 40th IEEE Conference on Decision and Control (Cat*. 3, vol. 3: No. 01CH37228), 2001, pp. 2899–2904.

[20] N. Vasilache, C. Bastoul, and A. Cohen. “Polyhedral code generation in the real world”. In: *Compiler Construction: 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006*. Heidelberg: Springer, 2006, pp. 185–201.

[21] Sven Verdoolaege. “isl: An Integer Set Library for the Polyhedral Model”. In: *Mathematical Software – ICMS 2010*. Ed. by David Hutchison et al. Vol. 6327. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302. ISBN: 978-3-642-15581-9 978-3-642-15582-6. DOI: 10.1007/978-3-642-15582-6_49. URL: http://link.springer.com/10.1007/978-3-642-15582-6_49 (visited on 07/10/2024).

[22] Sven Verdoolaege et al. “Polyhedral parallel code generation for CUDA”. en. In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), pp. 1–23. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2400682.2400713. URL: https://dl.acm.org/doi/10.1145/2400682.2400713 (visited on 07/10/2024).

[23] Weichuang Zhang et al. “An Optimizing Framework on MLIR for Efficient FPGA-based Accelerator Generation”. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Edinburgh, United Kingdom: IEEE, Mar. 2024, pp. 75–90. ISBN: 9798350393132. DOI: 10.1109/HPCA57654.2024.00017. URL: https://ieeexplore.ieee.org/document/10476481/ (visited on 09/20/2024).

[24] Y. Zhang and J. Yang. “Optimizing I/O for Big Array Analytics”. _eprint: abs/1204.6081. 2012.

[25] Ruizhe Zhao et al. “POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations”. In: *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. Belfast, United Kingdom: IEEE, Aug. 2022, pp. 235–242. ISBN: 978-1-66547-390-3. DOI: 10.1109/FPL57034.2022.00044. URL: https://ieeexplore.ieee.org/document/10035220/ (visited on 09/20/2024).

SACI Abdallah (photo not available) is a researcher in the Computer Science Department at Batna 2 University in Algeria. He received his master’s degree from Batna University in Algeria, with a specialization manufacturing scheduling. Currently, he holds the position of assistant professor. His research interests primarily revolve around code generation, with a specialization in Polyhedral model.

SEGHIR Rachid (photo not available) is a full professor at the university of Batna 2, Algeria. He received his engineering degree from the university of Batna in 2000, and DEA and PhD degrees in computer science from the university of Strasbourg, France in 2002 and 2006 respectively. His research interests include symbolic counting problems, program optimization and parallelization, polyhedral model, and parallel computing. He is also involved in some other research areas, such as applied mathematics and natural-inspired optimization.