Array Size Computation under Uniform Overlapping and Irregular Accesses

Angeliki Kritikakou, Francky Catthoor, Vasilios Kelefouras, Costas Goutis

Array Size Computation under Uniform Overlapping & Irregular Accesses

ANGELIKI KRITIKAKOU, IRISA & Dep. Computer Science & Electrical Engineering, Univ. of Rennes 1
FRANCKY CATTHOOR, IMEC & Dep. Electrical Engineering, Kath.Univ. Leuven
VASILIOS KELEFOURAS, Dep. Electrical & Computer Engineering, Univ. of Patras and
COSTAS GOUTIS, Dep. Electrical & Computer Engineering Univ. of Patras

The size required to store an array is crucial for an embedded system, as it affects the memory size, the energy per memory access and the overall system cost. Existing techniques for finding the minimum number of resources required to store an array are less efficient for codes with large loops and not regularly occurring memory accesses. They have to approximate the accessed parts of the array leading to overestimation of the required resources. Otherwise their exploration time is increased with an increase over the number of the different accessed parts of the array. We propose a methodology to compute the minimum resources required for storing an array which keeps the exploration time low and provides a near-optimal result for regularly and non-regularly occurring memory accesses and overlapping writes and reads.

Categories and Subject Descriptors: B.3.2 [**Memory structures**]: Design Styles—*Primary Memory*; C.3 [**Special purpose and Application-Based Systems**]: Real-time and Embedded Systems; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; E.1 [**Data Structures**]: Arrays

General Terms: Design

Additional Key Words and Phrases: Liveness, Resources Optimization, Near-optimality, Scalability, Iteration Space

1. INTRODUCTION

The size required to store the information in a system is crucial for several domains, because it is directly coupled with the cost of the system, the required area and the energy consumption [Catthoor 1999]. Such examples are the scratchpad memories of embedded systems [Grösslinger 2009], the hardware controlled caches of general purpose systems [Catthoor 1999] and the industry storage management systems, such as cargo [Lee et al. 2007]. In the embedded systems the memories have an important part in the overall system cost [Catthoor 1999]. The embedded applications, such as image, video and signal processing, mostly use arrays as data structures. Then, the size required to store the arrays becomes an essential part of the overall system design cost. An overestimated size leads to increase in the memory size, the chip area and the system energy consumption.

In this work we are computing the minimum size required to store an array under a given schedule, which is required in memory optimizations and more pre-

Author's address: A. Kritikakou, Dep. Computer Science & Electrical Engineering, Univ. of Rennes 1, & IRISA-INRIA 35000, Rennes, France, angeliki.kritikakou@irisa.fr, C. Goutis, V. Kelefouras: Univ. of Patras, Dep. Electrical & Computer Engineering, Rio, Patras, Greece, 26500, F. Catthoor: Inter-university Micro-Electronics Center, Kapeldreef 75, 3001 Leuven, Belgium

DOI:http://dx.doi.org/10.1145/0000000.0000000

cisely in the intra-signal in-place step of the Data Transfer and Storage Exploration (DTSE) [Catthoor et al. 1998]. The size is defined by the maximum number of array elements that are concurrently alive, i.e. all the elements that have been already stored (written) in memory and they will be loaded (read) in the future. When the writes and the reads to the array are overlapping, the complexity of the computation of the maximum number of alive elements is increased. In the overlapping case the reads of the array start before all the writes have finished. Therefore, the number of concurrently alive elements is not unique among iterations, as the lifetime of each element can be different. The overlapping case is different from the non-overlapping one [Kritikakou et al. 2013]. In the latter, all the reads to the array are performed after all the writes have finished and thus the maximum number of concurrently alive elements is the footprint of the array.

Existing techniques to compute the minimum size required to store an array are enumerative, symbolic/polyhedral or approximations, as described in Section 10. The enumerative approaches find the optimal size, i.e. the exact number of concurrently alive elements. However, when the array size is increased, the required exploration time to find the size is highly increased. The symbolic/polyhedral approaches [De Greef et al. 1997; Cohen et al. 1999; Darte et al. 2005; Clauss et al. 2011] use inequalities to describe the edges of the shapes that describe the accessed parts of the array. They provide the maximum number of alive elements by finding the maximum number of integers through an ILP solver. Hence, when the array size is increased, some values in the inequalities are modified, and thus, they are scaling well providing exact results. However, when the accesses to the array are not regularly occurring, i.e. they become irregular, the shape that describes an accessed part of the array loses its solidity, i.e. regions exist inside the shape which are not accessed, creating holes. In addition, the number of shapes of the accessed parts are increased and these shapes may not be regularly repeated in the space. This irregularity is created by the data dependent and the use case dependent array accesses. When the shapes of the accessed parts are not similar to polyhedra or lattices, the symbolic/polyhedral approaches lose their ability to efficiently describe them. The number of inequalities required to describe the edges of the accessed shapes is increased, increasing the time of the ILP solver. To reduce the number of inequalities and thus the exploration time, the shapes can be approximated, for instance using (approximated) convex hulls or approximated enumerators. However, no control exist over the quality of the approximation. An approximated convex hull can be applied to solidify both the holes inside and between the accessed parts in order to create a simple shape. However, this process can highly overestimate the number of concurrently alive elements leading to an overestimation of the size of the required resources.

The notation used in the rest of this paper is summarized in Table I. The paper organization is: Section 2 describes the motivation of our work providing relevant examples, Section 3 describes the contribution of the proposed methodology. Section 4 describes the target domain and the input to our approach, Section 5 described the translation of the access scheme into patterns and Section 6 describes the computation of the size. Section 7 and 8 presents the closed form solutions for several cases. Section 9 presents evaluation results. Section 10 presents the related work on techniques using the maximum number of concurrently alive elements. Section 11 concludes this study.

2. MOTIVATION

The high irregularity in the array accesses of the applications derives from aspects that cannot be modelled accurately statically and from different operation modes or

Table I: Summarized notation of the proposed methodology.

SIS	Solid Iteration Space
ISH	Iteration Space with Holes
ECS	Enumerative Conditions for SIS
ECH	Enumerative Conditions for ISH
PCS	Parametric Conditions for SIS
PCH	Parametric Conditions for ISH
SIR	Segment Iterator Range
ST	Segment Type
LB	Lower Bound
UB	Upper Bound
IR	Iterator Range of pattern (IR=UB-LB-1)
PS	Pattern Size ($PS = \sum_{i=0}^{\mathsf{N}} SIR(i)$)
RF	Repetition factor ($RF = \frac{IR}{PS}$)
N	Total number of pattern segments
Diff	Exploration window, i.e. index difference of RD and WR
SID(i)	Segment Iterator Domain of i pattern segment
HID(i)	Hole Iterator Domain of i pattern segment
Size_i	Size of i dimension
$\mathsf{PSize}_\mathsf{i}$	Partial size of exploration window of i dimension
$\mathsf{Asize}_\mathsf{i}$	Additional elements of exploration window of i dimension
CP	Combined Pattern
ADiff	Maximum number of A in the pattern section
HDiff	Maximum number of H in the pattern section
$extra_{\mathrm{PCH}_=}$	Additional elements due to a PCH of $==$ type
$extra_{\mathrm{PCH}_=,\mathrm{A2A}}$	Additional elements due to combination of PCHs of $==$ type
$extra_{\mathrm{PCH}_\neq}$	Additional elements due to PCH of $\neq$ type
$extra_{\mathrm{WR}}$	Computation of additional written elements
$extra_{\mathrm{RD,A}}$	Computation of additional not yet read elements before the exploration window where the WR statement is A
$extra_{\mathrm{RD,H}}$	Computation of additional not yet read elements before the exploration window where the WR statement is H

use cases of the application. For instance, different video decoder modes like I, P and B frames with different submodes [Juurlink et al. 2012] and applications with data dependent conditions and loop bounds.

To deal with the different use cases or to remove the data dependencies in order to continue with an efficient Design Space Exploration (DSE), the system scenarios approach [Filippopoulos et al. 2014; Filippopoulos et al. 2012; V.Gheorghita et al. 2009] is a very promising direction. The alternative of considering the general worst case of all the modes and the code instances leads to pessimistic over-approximations. The system scenarios approach explores the potential values of the data and their effect in the application. Based on this information it groups similar instances of the application code into a common scenario. Due to the initial data dependencies the resulting application instances have highly irregular array accesses. Now, we can use the worst case code instance per scenario to continue with the DSE. The result of the system scenario approach is a number of scenarios that have to be explored, where each scenario has a different code instance with irregular array accesses. The number of code instances to be explored is equal to the number of scenarios. A trade-off exists between the number of scenarios and the similarity in the code instances. In this paper we do not deal with the system scenario creation, but we use the resulting codes as an input to our methodology. More information about the memory-aware system scenario approaches is presented in [Filippopoulos et al. 2014].

As we now have to explore a number of different application instances in this DSE step, a general parametric approach is not applicable. The time required to explore one instance becomes now an important factor in the overall time of DSE, as we need to apply this step a lot of times in order to explore a set of different scenarios for the application under study and we have to apply this process for all the arrays existing in the application. To provide an intuition over the gains in the overall flow, we will use

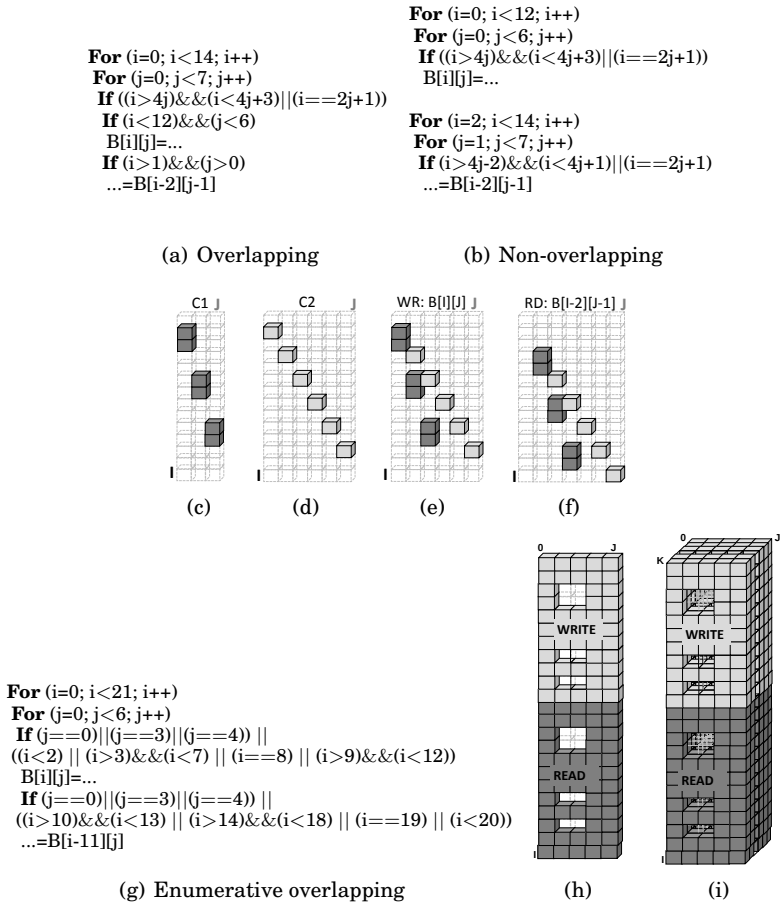
the information provided in [Balasa et al. 2008] using the STOREQ, a tool developed for main parts of the steps [Kjeldsberg et al. 2003]. This tool is typically used in the inner core of a loop transformation exploration kernel. The number of different loop organisation schemes that have to be explored can be huge, e.g. for permutations it is exponential and by adding loop splits this number is prohibitively increased. For instance, for the Darwin algorithm with 27 arrays we observe the longest time for the overall flow, i.e. 2.5s [Balasa et al. 2008]. Assuming that the array size computation for polyhedral approach takes 81 ms, which was the lower value obtained during our experimentations, then the total time dedicated to the computation of the maximum alive elements for all the arrays is 2.187s, which is the dominant task as it takes 88% of the total time. Replacing this step with the proposed approach (which in the worst observed case takes 0.902 ms), we need 24.36ms for this step (a gain factor of 89.77) and the total time is 337.36ms (gain factor of 7.4). As the computation of the maximum number of alive elements is one step in the overall DSE process, we should keep the exploration time reasonable, while providing a near-optimal result.

The near-optimality of this DSE step is essential, as the obtained result is used in the next memory DSE phases, i.e. inter-signal in-place optimization, memory access scheduling and memory address generation [Catthoor 1999], and thus the quality of the result of this step affects the quality of the next phases. Therefore, we do not want to increase the memory size without a reason.

As a high level of irregularity exists in the array accesses inside each code instance, the existing approaches either require a higher exploration time to find an near-optimal solution or they have to approximate the result. The enumerative approaches add the writes between the iteration where an element is written and the iteration where this element is read for the last time. This process is repeated for all the elements of the array. Hence, when the number of the accesses is increased, they require significant exploration time. The symbolic/polyhedral approaches can represent the irregular space as unions of polytopes. However, when the number of different polytopes is increased, the exploration time is also increased. The alternative is to approximate the array accesses by applying (approximated) convex hulls which can over-approximate the size as the approximation is not performed in a controlled way. The proposed methodology remains scalable and near-optimal, as it represents the array accesses as patterns with accesses and holes and it combines iteratively the patterns to find the required size. In case long patterns occur, the knowledge of the holes allows us to approximate in a controlled way. By selecting few small holes to be considered as accesses, the size of the patterns and the time of the proposed approach are reduced. More details are provided in the rest of the manuscript.

2.1. Examples

Real-life applications are characterized by dynamic behavior which is also reflected in fluctuating memory requirements [Filippopoulos et al. 2014]. Then, the conventional allocation and assignment of data remains suboptimal. One example of such a code is Hough transformation [Duda and Hart 1972], where the element of the accumulator array that is accessed each time depends on the value of the image pixel and its position. Therefore, the accesses to the accumulator array are highly irregular without following any specific pattern. In a conventional assignment the memory size used for the storage of the accumulator is determined by the dimensions of the input image using the worst-case area for the accumulator array variable. However, only a part of the allocated space may be accessed. To improve the memory management, memory-aware system scenario approaches are applied [Filippopoulos et al. 2014], which create a set of scenarios with different code instances describing highly irregular accesses to the accumulator.



(a) Overlapping (b) Non-overlapping

(c) (d) (e) (f)

(g) Enumerative overlapping (h) (i)

Fig. 1: Application code for (a) overlapping and (b) non-overlapping case, iteration spaces where the boxes describe accesses of the (c) C1, (d) C2, (e) written elements, (f) read elements, (g) overlapping application code with enumerative conditions, (h) iteration space with accesses and (i) extension to 3 dimensional case.

In the examples of the paper we keep the iteration space small for demonstration reasons. Fig. 1 explores how the existing approaches behave on the different code instances with high irregularity derived from the memory-aware scenario. The overlapping application code is depicted in Fig. 1(a) and the non-overlapping code in Fig. 1(b) to illustrate the difference between these two cases. The non-overlapping code consists of two nested loops. The first loop performs a WR statement B[i][j] with conditions C1: $(i > 4j)\&\&(i < 4j + 3)$ and C2: $i == 2j + 1$ combined with an OR operation. The second loop performs a RD statement B[i-2][j-1] with conditions C1: $(i > 4j - 2)\&\&(i < 4j + 1)$ and C2: $i == 2j + 1$ combined with an OR operation. The index difference of the RD and the WR statement is 2 in I dimension and 1 in J dimension. The overlapping code consists of one nested loop with the same conditions and some additional ones which describe the beginning and the end of the iteration space of the WR and RD statements. In the overlapping case at least in one iteration the WR and the RD statements are both executed. For instance, we observe that when I=3 and J=1 both a WR and a

RD occurs to array B. The element B[1][0] is read and the element B[3][1] is written, as depicted by the corresponding box in the iteration I=3 and J=1 at Fig.1(e) and Fig.1(f). For this iteration, the number of alive elements is 2. For a later iteration (I=6, J=1), the element B[6][1] is written, but no read occurs to free a memory position increasing the number of alive elements to 3. In contrast, in the non-overlapping case during the iterations where the writes are executed, no read occurs at the array. Therefore, by applying the solutions of the non-overlapping case [Kritikakou et al. 2013], the number of alive elements will be computed equal to 11.

When only one of the conditions C1 or C2 exists, the access shape consists of holes repeated in a regular way (Fig. 1(c) and Fig. 1(d)). This regular access scheme is efficiently represented by the symbolic/polyhedral approaches [Franssen et al. 1993; Darte et al. 2005; Clauss et al. 2011]. When both conditions coexist, the repetition of the first access shape C1 is disturbed due to the second access shape C2. In this example, the symbolic/polyhedral approaches represent the space as the union between these two polytopes. For the example of Fig. 1(a), the exploration time is 627 ms for the polyhedral approach [Verdoolaege et al. 2013] and 27.8 sec for the enumerative approach to find the optimal result. However, when the number of different polytopes is increased, the exploration time is also increased, as shown in the experimental results. Another example is provided in Fig. 1(h) (code of 1(g)) which depicts the access shape of a WR for a two dimensional case. The symbolic/polyhedral approaches need at least 6 polytopes to describe the access shape when intersection and union are used in the representation and 3 when subtraction is also included. The more irregularity is present, the more polytopes are required to describe the space. The exploration time is 862 ms for the polyhedral approach and 38.8 sec for the enumerative approach to find the optimal result. The alternative is to approximate the accessed region by a convex hull. In this example, the holes of size 2x2 and 1x2 are considered as accesses and the computed size is 55, instead of 47, and thus a loss of 17% is observed. By adding another dimension, as depicted in Fig.1(i), the loss due to the convex hull approximation, which is 8 in our example, is increased, as it is multiplied by the size of the dimension K. The exploration time is 1482 ms for the polyhedral approach and 1223.5 sec for the enumerative approach.

2.2. Target domain & Problem formulation

The domain under study consists of applications of one thread with high memory activity and with and without varying memory access intensity [Filippopoulos et al. 2014], which possibly depends on input data variables. Such examples are high-speed data intensive applications in the fields of speech, image and video processing, which require significant amount of storage resources [Jha et al. 1997]. In the case of varying memory activity, a static study of the application code is insufficient since the applications have non-deterministic behaviour and thus memory-aware systems scenario approaches are applied to remove the dynamic behavior by creating a number of scenarios [Filippopoulos et al. 2014]. Each scenario has different code instances with deterministic behaviour and high irregularity.

Therefore, the proposed methodology is applied to applications that have static control flow and regular and irregular array accesses. The irregularity derives from applications with data dependent accesses to the arrays through non-manifest conditions and non statically known loop bounds. Applications with static nested loops and manifest conditions, i.e. they can be analyzed to deduce which values they take without executing the program, can also destroy the regularity of array accesses but in a lower degree than the data dependent accesses. After applying the system scenario approach [Filippopoulos et al. 2014; V.Gheorghita et al. 2009], we obtain a set of codes instances of the initial application with static nested loops and manifest affine condi-

tions, which describe different and irregular accesses to the arrays. Then, the worst case code instance is explored to find the minimum required size to store the arrays. However, in this work we deal with the computation of the maximum number of alive elements, whereas the creation of systems scenarios is described in [Filippopoulos et al. 2014].

Our approach focuses on the computation of the size of an array, which belongs to the intra-signal in-place optimization step of the DTSE. The DTSE methodology [Catthoor et al. 1998] has three phases: (1) the pruning phase which preprocesses the program to enable the optimization phase, (2) the DTSE methodology that optimizes the preprocessed program and derives a memory hierarchy and layout, and (3) the depruning and address optimization to remove the overhead introduced by DTSE. During the pruning phase the Dynamic Single Assignment (DSA) form is used to represent the applications in order to enable the memory optimizations. Several techniques exist to write the application in DSA form, such as the work in [Feautrier 1988; Vanbroekhoven et al. 2003; Vanbroekhoven et al. 2005]. The code application for our methodology writes at most once each array element during the execution of the program, but it can read it several times [Vanbroekhoven et al. 2005]. An example of transforming to DSA form is the DGEMM of two two-dimensional arrays, where C[N][N] becomes C[N][N][N] and the 3rd dimension is used to index the values produced by the k loop. When we consider the complete application and array C is used later by another part of the program, then the number of maximum alive elements we compute using our methodology is N^2. Potential storage overhead introduced by the DSA due to the elimination of the write-after-write dependencies can be removed as the in-place mapping intentionally destroys the DSA property [Vanbroekhoven et al. 2007]. The result of our methodology provides the minimal array size for the DSA. The values of the array elements that are no longer needed can be overwritten by changing the indexation of the assignments [Eddy De Greef and de Man 1996; Tronon et al. 2002].

After the memory management is completed by DTSE, all the required information is available to execute the address optimization and the final mapping/allocation [Catthoor et al. 1998]. The proposed approach supports an efficient address scheme achieved when the addresses are quite regular, as it does not allow exhaustive enumeration of accesses and uses repetitive patterns. When the size of patterns is increased, the proposed methodology provides a control to reduce it by considering holes in the accesses with low repetition factor and of small size as virtual accesses. In this way, the address complexity is reduced.

3. CONTRIBUTION

The contribution of this work is to propose a methodology which computes the maximum number of concurrently alive array elements keeping the exploration time low and providing a controlled approximation when required, resulting to a near-optimal and scalable approach. Compared to existing approaches, the proposed methodology achieves low exploration time and near-optimal results in complex iteration spaces with both regular and irregular array accesses and overlapping writes and reads, as also verified by the experimental results in the Section 9.

The proposed methodology consists of three steps as depicted in Fig.2: the analysis, the translation and the computation. When the methodology is applied, the steps are executed following a sequence and the information is propagated from the analysis to the computation step.

The analysis extracts the access scheme from the application code. As described in Section 4, the input code is described by a parametric template. By parsing the code under study, the template is instantiated. In this way we extract the access scheme, which is propagated to the translation step.

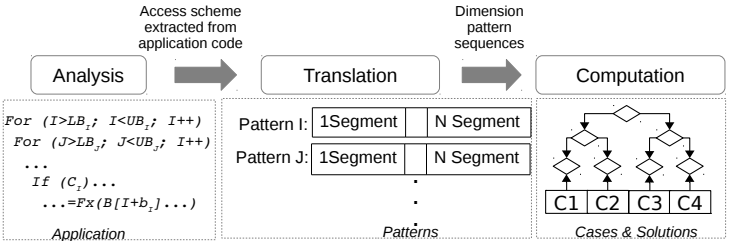


Fig. 2: Methodology steps.

The translation step stores the access scheme received by the analysis step in a near-optimal and scalable way using the representation of the pattern formulation presented in [Kritikakou et al. 2014]. The patterns and the pattern operations are applied per dimension. A pattern is a sequence of segments that describe Accesses (A) or Holes (H), i.e. not accesses, to the array and has a set of parameters that describe the position and the repetition of the pattern in the iteration space. In case more than one pattern refers to the same parts of the iteration space, pattern operations are applied to combine these patterns [Kritikakou et al. 2014], whenever it is possible. The obtained result is a sequence of patterns for each dimension. The pattern sequences are propagated to the computation step.

The focus of this paper is the last step, i.e. the computation step, which takes as input the pattern sequences provided by the translation step. In contrast to the work presented in [Kritikakou et al. 2014], the proposed methodology describes a framework to compute the maximum number of alive elements for regular and irregular access schemes for overlapping writes and reads. The proposed framework describes the possible cases that may occur by combining the pattern sequences of the translation step. We have applied a systematic approach to derive the complete set of the potential cases where we divide the exploration space in two non-overlapping and complementary parts, inspired by the approach presented in [Kritikakou et al. 2013]. In addition, for each case we define a set of closed form equations and/or low complexity algorithms using parameters. During the execution of the computation step, the parameters are instantiated, i.e. they take specific values, based on the application under study each time. The parameter instantiation decides which cases are valid from the framework and thus selects the solutions to be applied.

To motivate the need of the proposed methodology we describe the gains of the motivational examples of Fig.1. The proposed approach translates the extracted C1 condition of Fig.1(a) to the pattern {1H 2A 1H} with a repetition factor equal to 3 and the C2 condition to {1H 1A} with a repetition factor equal to 6. At computation step, the size is computed equal to 3. The exploration time of our method is 103 ms. For the Fig.1(h), the pattern of I dimension is {2A 2H 3A 1H 1A 1H 1A} and the one of J dimension is {1A 2H 2A}. The size is computed by propagating the partial size due to I dimension (outer dimension) to J dimension (inner dimension). The partial size is given by the accesses of I dimension multiplied by the loop size of J dimension, i.e. 7*5. Then, the additional accessed elements in J dimension are added without including the accesses already computed in I dimension. Hence, the size in J dimension is given by the holes of I dimension multiplied by the accesses of J dimension, i.e. 4*3. The result is 47, which is optimal, and the exploration time is 82 ms.

Pseudocode 1: Loop structure of the application code instances

```
for (i=LB_i;i < UB_i;i++) do
    for (k=LB_k;k < UB_k;k++) do
        if (C^1(i, type_i^1, a_i^1)..) then
            if (C^n(k, type_k^n, a_k^n)..) then
                S^1:A[i+b_i^1]..[k+b_k^1]=....
        if (C^1(j, type_j^1, a_j^1)..) then
            if (C^2(l, type_l^1, a_l^1)..) then
                S^2:...=F_2(A[i+b_i^2]..[k+b_k^2])
```

4. ANALYSIS STEP

The input to our methodology is the code of the application under study. We present a parametric template with primitive components to describe the structure of the application code.

The analysis template consists of a generic structure depicted in Fig. 1 that describes the application using a nested loop with 1) a set of condition expressions described by primitive conditions and primitive logical operations and 2) single write multiple read access statements that use the primitive index expression. The write and read accesses are overlapping, i.e. at least in one iteration both write and read statements are executed, and thus the lifetime of the array elements is not unique. This case is complementary to the non-overlapping case [Kritikakou et al. 2013] and together they cover all possible options that may occur.

In this paper, we will consider this type of applications as input to describe the solutions of the proposed methodology. The approach can be further extended by generalizing the description of the codes that are given as input to the methodology. However, this is not the focus of this paper. We provide several hints for the direction of the transformations that have to be applied. The use of primitive conditions and logical operators provides a uniform input to our methodology and avoids the exploration of the different ways of writing an application code.

The primitive conditions are expressed through manifest control affine statements, C^X(iterator, comparison-operator, expression) which describes a condition of "iterator comparison-operator expression", e.g. $C^1(i, <, 5)$ describes i<5 condition. The array access statements are described by the function $Fx(A[f_{\mathrm{x}}^{iterator_1}]..[f_{\mathrm{x}}^{iterator_{A_{\mathrm{Dim}}}}]))$. Fx is a function of accessing arrays, f_{x} is the index expression for the iterator of each dimension, which describes a set of array accesses to the background memory, A_{Dim} is the number of array dimensions and A is the array being accessed. The array access statements are uniformly generated, i.e. the index expressions of the array accesses differ in a constant term. The access statements allow both group and self reuse since two different access statements can read the same element and an access statement can read more than once a given element. Array A is used as an example to describe the arrays of the application domain and many arrays similar to A may exist in the application, for which the methodology is repeated. As we are focusing on loop dominated applications, few condition statements exist inside the loops and a high kernel unrolling cannot occur. Hence, the number of condition and access statements in the application is a small finite set. As the loop dimensions are few, the information extracted and used by our approach is also a finite small set.

Definition 4.1. The *types of iteration spaces* are

(1) *Solid Iteration Space (SIS)*, i.e. iteration space where the accesses to the array occur without holes,

(2) *Iteration Space with Holes (ISH)*, i.e. iteration space where the accesses to the array occur with holes.

For instance, the presented motivational examples describe ISH.

Definition 4.2. The *nested loop* is described by the
(1) *Dimensions*: The number of nested levels,
(2) *Order*: The loop iterators order from outer to inner dimensions,
(3) *Dimension description*: For each dimension Dim we have the lower bound LB, the upper bound UB and the step ST, which are given by numerical values and not in a parameterized form.

For instance, the description of the nested loop of the example presented in Fig. 1(a) has two dimensions which have the order i,j. The description of i dimension is LB=-1, UB=14 and ST=1 and the one of j dimension is LB=-1, UB=7 and ST=1.

Definition 4.3. The *types of conditions* are *Enumerative*, where the expression in the condition is only a constant, and *Parametric*, where the expression in the condition is an affine expression.

For instance, in the description of the example of Fig. 1(a) two parametric conditions exist, (i>4j)&&(i<4j+3) and (i==2j+1), and four enumerative conditions exist, two for the write statement, (i<12)&&(j<6), and two for the read statement, (i>1)&&(j>0). Further enumerative conditions are given in the example of Fig. 1(g), e.g. (j==0), (j==3), (i>3)&&(i<7), (i>14)&&(i<18) etc.

Combining the types of the iteration spaces and the types of the conditions we derive the primitive conditions which we use in our parametric template and can describe all the possible cases that can occur in the applications.

Definition 4.4. The *type of primitive conditions* COND of the analysis template is:
(1) *Enumerative Conditions for Solid iteration space (ECS)*, which are expressed through the $<$ and the $>$ comparison operators, the combination of the $<$ and the $>$ comparison operators with the AND logic operator ($<$ && $>$) and the $==$ comparison operator, i.e. $i < a$, $i > a$, $(i < a)\&\&(i > b)$ and $i == a$.
(2) *Enumerative Conditions with Holes (ECH)*, which are expressed by ECS combined with || primitive operator.
(3) *Parametric Conditions for Solid iteration space (PCS)*, which are expressed through the $<$ or $>$ comparison operator with a linear expression, i.e. $i < c * k + d$ and $i > c * k + d$.
(4) *Parametric Conditions for Iteration Space with Holes (PCH)*, which are expressed with the $==$ or $\neq$ comparison operator, a combination of $<$ and $>$ comparison operator with an && logic operator, i.e. $i == c * k + d$, $i \neq c * k + d$ and $(i > c * l + d_1)\&\&(i < c * k + d_2)$ (with $d_1 < c$ and $d_2 < c$).

For instance an ECS is $i < 5$, an ECH is $i == 5$, a PCS is $i < 2 * k + 1$ and a PCH is $i == 4 * k + 1$. In our template we will use the comparison operators of $<$, $>$, $==$, and $\neq$.

Definition 4.5. The *primitive condition operations* of the analysis template are the OR (||) and the AND (&&).

Definition 4.6. The *primitive index expression* which describes f_x of the access statement is the expression $iterator + constant$. We consider that the default index expression of the WR statement as i and of the RD statement as $i + b$.

The primitive index expression is a highly occurring case, especially in multimedia applications, e.g. [Pouchet et al. 2012; Lee et al. 1997; Guthaus et al. 2001].

In case the application under study has been written in a different way that the generic one provided by the analysis template, a set of preprocessing operations are applied to map the application code to our generic structure. The applied operations are not transformations to the original code, as they are only applied for the mapping to the analysis template. In case of:

(1) Non-primitive logical operators: they are transformed to combinations of the primitive operations. For instance, $i == a$ NAND $j == b$ is transformed to $(i \neq a)||(j \neq b)$.
(2) Non-primitive conditions: they are mapped to primitive conditions. For instance, the ECH condition $i \neq d$ is mapped to the condition $(i < d)||(i > d)$.
(3) The WR index expression is not the default (i): It is mapped to the default, e.g. when the WR index expression is $i + a$ and the RD index expression is $i + b$, the index expressions are translated to i and $i + (b - a)$, respectively.

In case that different index expressions than the primitive one are used, we provide a set of possible preprocessing than can be applied to map them into the primitive index expression. However, in this paper we are focusing on presenting the proposed methodology using the primitive index expression.

(1) Index expressions with one iterator: These cases are similar to the primitive index expression:
   (a) The WR index expression is "coefficient iterator + constant" (e.g. $ai + b$): If the coefficient is the same for the WR and RD, then the methodology can be used directly, as the patterns of the access statement have constant difference. In case different coefficients are used, one option is to translate every statement into "iterator + constant" index type with a new loop dimension and parametric manifest conditions to describe the holes in the iteration space. For instance, index expression $2 * i + 1$ executed from 0 to 9 is expressed as index expression $I + 1$ and an additional loop dimension K from 0 to 4 and a parametric condition $I == 2K$.
   (b) The WR index expression is "iterator $\%$ constant" (e.g. $i\%a + b$): This iterator can be expressed as "iterator + constant" index type and an additional loop dimension with size equal to the constant. For instance, the index expression $i\%4$ becomes K and creates an K loop dimension from 0 to 3.
(2) Index expressions which couple iterators: Index expressions such as $i + j + b$ are less similar with the primitive index expression by the current version of the proposed methodology. Heuristics to replace the i + j + b by a new iterator I and taking the worst case in terms of bounds could be applied, but we are not focusing on this case in this version of our methodology. We have as future work to provide further extensions to the proposed methodology to near-optimally cover the case of index expressions with coupling iterators.
(3) Non-affine index expressions: Existing work on transformations of complex expressions to simpler ones, e.g. [Paek et al. 2002] could be applied to create approximated affine expressions.

In case multiple write statements exist and the writes and the reads are executed:

(1) sequentially, i.e. a write statement and then a set of reads of this written element, our methodology is applied for each pair of a single write and several reads.
(2) interleaved, they can be merged into a single statement combined with conditions which control the correct behavior of the write statement.

Under these assumptions, the proposed approach is widely applicable. For instance, by analysing the different codes provided by the Polybench benchmark suite (28 different codes), the proposed approach is applicable in 23 codes, that is 82,14%. The reasons for which our approach is not applicable in the current form for the remaining code are the non-uniform array accesses statements and the coupling of iterators.

In the remaining sections, we illustrate the solutions using single read statements. When multiple reads exist, the last read of an element is used as it defines the element's lifetime.

As the gains of our approach are in iteration spaces with holes, we focus on the conditions that are able to create such iteration spaces, i.e. ECH and PCH primitive conditions. In addition, we describe also the ECS which describes solid iteration spaces in order to provide solutions in the cases where solid spaces are combined with spaces with holes. The condition PCS describes solid iteration spaces similar to the ECS, but they are parametric. We leave as future work the solutions for the PCS, as existing approaches such as polyhedra provide exact results for this regular case [Clauss et al. 2011]. Our approach can be extended to PCS, as they can be computed in a similar way to ECS with the number of accesses elements derived by the exploration window each time.

5. TRANSLATION STEP

The translation receives the extracted information from the analysis and stores it in patterns and applies pattern operations to combine the patterns per dimension.

Definition 5.1. A *pattern* is defined by a sequence of segments, a type and a set of parameters. A segment is described by:

(1) the Segment Type (ST), i.e. the type of behavior of the statement. It can be Access (A) or Hole (H),
(2) the Segment Iterator Range (SIR), i.e. the number of consecutive iterator values where the statement has the same segment type:
   (a) if ST $==$ A of a segment i, it is called Segment Iterator Domain (SID(i)).
   (b) otherwise, it is called Hole Iterator Domain (HID(i)).

The type of the pattern is a RD, WR or COND, where the condition type is given by Definition 4.4.

The set of parameters are:

(1) the Lower Bound (LB), is the iterator value before the pattern begins. For the ECS/PCS and ECH conditions ($I > a$) the LB is $max(\mathsf{LB}_\mathrm{I}, a)$ and for PCH conditions ($I > c * K + d$), is $max(\mathsf{LB}_\mathrm{I}, c * (\mathsf{LB}_\mathrm{K} + 1) + d - 1)$,
(2) the Upper Bound (UB), is the iterator after the pattern ends. For the ECS/PCS and ECH conditions ($I < b$) the UB is $min(\mathsf{UB}_\mathrm{I}, b)$ and for the PCH conditions ($I < c * K + d$) is $min(\mathsf{UB}_\mathrm{I}, c * (\mathsf{UB}_\mathrm{K} - 1) + c + 1)$.
(3) the Iterator Range (IR), i.e. the iterator values where the pattern is valid, IR=UB-LB-1
(4) the Number of pattern Segments (N), i.e. the total number of segments in the pattern,
(5) the Pattern Size (PS), i.e. the summation of the SIR of all the segments in a pattern, i.e. $\sum_{i=1}^{\mathsf{N}} \mathsf{SIR}(i)$.
(6) the Repetition factor (R), which describes how many times the pattern is repeated in the IR, i.e. $\frac{\mathsf{IR}}{\mathsf{PS}}$.
(7) the exploration window (Diff), which is defined for the patterns of WR and the RD type and describes the difference between the RD and the WR index expressions.

We use the examples presented in Section 2 to illustrate how the patterns are defined and how the pattern operations are applied. In the examples of Fig. 1(a), for the first parametric condition, the pattern derived for the dimension i is {1H 2A 1H}. The type of the pattern is COND and more precisely PCH. The corresponding parameters are LB is -1, UB is 14, IR is 14, N is 3, PS is 4 and R is 3. For the second parametric condition, the pattern derived for the dimension i is {1H 1A} with LB is -1, UB is 14, IR is 14, N is 3,

PS is 2 and R is 6. The first enumerative condition gives a pattern for the i dimension, which is {12A}. The type of the pattern is ECS. The corresponding parameters are LB is -1, UB is 12, IR is 12, N is 1, PS is 12 and R is 1. The second enumerative condition gives a pattern for the j dimension, which is {6A}. The type of the pattern is ECS. The corresponding parameters are LB is -1, UB is 6, IR is 6, N is 1, PS is 6 and R is 1. The access statement gives two patterns, one for each dimension. For the i dimension it is {14A}. The type of the pattern is WR. The corresponding parameters are LB is -1, UB is 14, IR is 14, N is 1, PS is 14, and Diff is 2. For the j dimension it is {6A}. The type of the pattern is WR. The corresponding parameters are LB is -1, UB is 6, IR is 6, N is 1, PS is 6, R is 1 and Diff is 1. Similar are the patterns for the other enumerative conditions and the read statement.

The algorithm of pseudocode 2 describes how we select the pattern operations between two patterns P1 and P2 of a dimension i. This process is repeated for each dimension starting from the patterns with the lower LB and the larger size. Further information is provided in [Kritikakou et al. 2014].

Pseudocode 2: Selecting pattern operations.

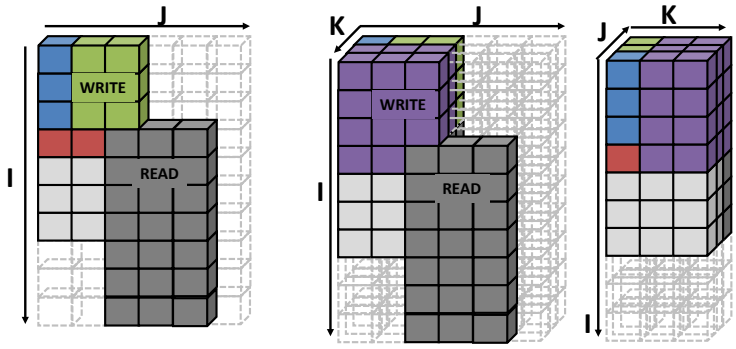
```
if (PS(P1) ≠ PS(P2)) then
    Modify patterns(P1,P2)
else
    if (LB(P1)+IR(P1) > LB(P2)) then
        if (LB(P1)+IR(P1) ≠ LB(P2) + IR(P2)) then
            if (LB(P1) < LB(P2)) then
                LB alignment(P1,P2)
            if (UB(P1) ≠ UB(P2)) then
                UB alignment(P1,P2)
        if (LB(P1)+IR(P1)==LB(P2)+IR(P2)) then
            PCH=Fully aligned operation(P1,P2, operation)
if (LB(P1)+IR(P1)==LB(P2)) then
    Sequential(PCH1,PCH2)
if (LB(P1)+IR(P1) < LB(P2)) then
    Non-sequential(P1,P2)
```

Using the pattern operations of Alg. 2 we combine the patterns to obtain the pattern sequence. For i dimension, we apply the UB alignment operation and then the fully aligned AND operation between the enumerative condition and the access statement. The result gives us the following pattern: {12A}. The type of the pattern is WR. The corresponding parameters are LB is -1, UB is 12, IR is 12, N is 1, PS is 12, and Diff is 2. Then, we apply a fully aligned AND operation with the first parametric pattern. The result is: {1H 2A 1H}. The type of the pattern is WR. LB is -1, UB is 12, IR is 12, N is 3, PS is 4, R is 3, and Diff is 2. Similarly, the result after applying the fully aligned AND operation with the second parametric condition is {1H 1A}. The type of the pattern is WR. The parameters are LB is -1, UB is 12, IR is 12, N is 3, PS is 2, and Diff is 2.

6. COMPUTATION STEP

The computation step is a framework with the different cases that can occur when the pattern sequences of different dimensions are combined. To find the different cases we explore the position cases, which derive from the position of the access statements in the application code and their behavior, and the condition cases, which derive from the loop structure of the code and the type of the conditions. Then, we propose an equation to compute the maximum number of alive elements per case. The creation of these equations derives by following an incremental computation of the size by adding dimensions.

Definition 6.1 (*Incremental size computation*). The incremental size computation:



(a) (b)

Fig. 3: General process to calculate the number of alive elements for a) two dimensions and b) three dimensions.

(1) starts by computing the partial size of the elements required to be stored due to the outer dimension, PSize_0;
(2) at each iteration i the size of an inner dimension is added by multiplying the size computed up to now Size_{i-1} with the size of the inner dimension Size_i and adding the additional elements required to be stored due to the new dimension ASize_i.

$$\mathsf{Size}_0 = \mathsf{PSize}_0 \quad (1)$$

$$\mathsf{Size}_{0-i} = \mathsf{Size}_{i-1} * \mathsf{Size}_i + \mathsf{ASize}_i, \text{for } i\text{=}1...\mathrm{Dim} \quad (2)$$

This general process is schematically illustrated in Fig. 3, where we assume that three dimensions exist, so iteration 0 corresponds to the outer dimension I, iteration 1 corresponds to the inner dimension J and iteration 2 to the second inner dimension K. The PSize_0 is computed by the required size only in the outer dimension I, i.e. the blue boxes in Fig. 3(a). These elements are the maximum number of elements that have been written and not yet read in the outer dimension. In the iteration 1, this size is extended by the size of the inner dimension J, Size_1, i.e. the green boxes in Fig. 3(a), and by the additional elements ASize_1 due to the inner dimension J, i.e. the red boxes in Fig. 3(a). In the second iteration, this size is extended by the size of the next inner dimension K, Size_2, i.e. the purple boxes in Fig. 3(b). In Fig. 3(b) we can see the iteration space using two different perspectives.

We describe the cases of our framework by presenting the position cases and the condition cases that can occur. Then, the remaining sections will describe how this general process is instantiated to create the final equations for each case and illustrate the solution using representative examples. In Table II, we provide a summary of the instantiations of the different cases, whereas Table IV provides the final equations for several representative examples.

6.1. Position cases

To define the different position cases that may occur, we need to explore the possibilities in the access statements and the loop structure. We apply a systematic approach where we divide the exploration space into two complementary and non-overlapping cases [Kritikakou et al. 2013]. By applying this process we obtain the splits depicted in Fig. 4 and described below.

Table II: Summary of instantiation of the general method based on the different cases.

	Dominant segment		**Non-dominant segment**	
PSize_0	Diff_0		ADiff_0	
	Dominant conditions			
Size_i	$\sum_{l=0}^{N_i} \text{SID}_i(l)$			
	Outer dimension			
	Dominant	**Non-dominant**	**Dominant**	**Non-dominant**
ASize_i	$\sum_{l=0}^{\text{Diff}_i} \text{SID}_i(l)$	0	$\sum_{l=0}^{\text{Diff}_i} \text{SID}_i(l)$	0
	Non-dominant conditions			
Size_i	IR_i		for ADiff_0: IR_i & for HDiff_0: $\sum_{l=0}^{N_i} \text{SID}_i(l)$	
	Outer dimension			
	Dominant	**Non-dominant**	**Dominant**	**Non-dominant**
ASize_i	Diff_i	$max(\text{extra}_{\text{WR}}, \text{extra}_{\text{RD}})$	Diff_i	$max(\text{extra}_{\text{WR}}, \text{extra}_{\text{RD}})$

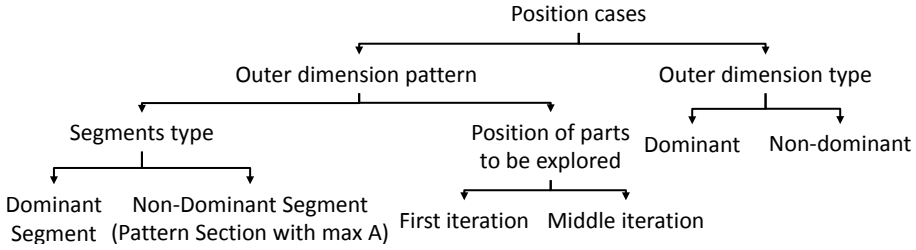


Fig. 4: Position cases.

The first division derives from the description of the pattern existing in the outer dimension and the type of the dimension. In the left part, the description of the pattern is divided into the type of the segments and the position of the segments to be explored. The type of the segments is described by the existence or not of a dominant segment in the outer dimension of the nested loop under study. When a segment of the pattern of the outer dimension that is accessed is larger than the exploration window of the outer dimension, then analyzing only this segment is enough to find the maximum number of alive elements of this dimension. Hence, this case is the **dominant segment**. On the other hand, if all the segments of the pattern of the outer dimension that are accessed are smaller than the exploration window of the outer dimension, we are referring to the **non-dominant segment**. In this case, the maximum number of concurrent alive elements cannot be found by exploring only one accessed segment, and thus we have to find the section that includes the maximum number of accesses in the pattern. Regarding the position of the segments to be explored (i.e. the aforementioned dominant segment or pattern section), they can be placed in the **first iteration** or in a **middle iteration** of the outer dimension. Exploring now the options in the type of the outer dimension, it can be dominant or not. The **dominant outer dimension** occurs when the behavior of the WR statement in the iteration just after the outer exploration window is equal to A. Because of this A and the behavior of the inner dimension, the number of alive elements is potentially increased. For instance, when a dominant segment occurs in the outer dimension and it is larger than its exploration window, then the outer dimension is dominant. In the case when the dominant segment is equal to

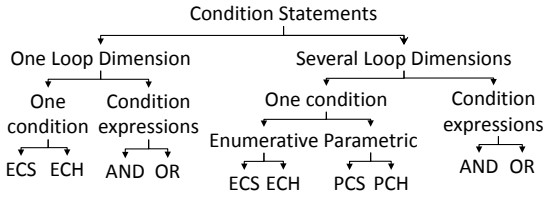


Fig. 5: Condition cases.

the outer exploration window, then the **outer dimension is non-dominant**, because an H exists in the WR statement in the iteration just after the exploration window.

6.2. Condition cases

By combining the different options of the loop structure, of the primitive conditions and the primitive logical operations that can occur in the generic structure of the analysis template, we define the possible condition cases for the condition statements depicted in Fig. 5. The first division is between a generic structure of a loop with one dimension and a structure with several loop dimensions. In the case of one dimension, a condition statement can be expressed by a primitive condition or by condition expressions with several primitive conditions. In the case of a primitive condition and one dimension, it can only be of enumerative type, i.e. ECS and/or ECH. In the case of several condition expressions and one dimension, the condition statement can combine the enumerative primitive conditions using AND and/or OR primitive operations. For several loop dimensions, a condition statement can be a primitive condition or several condition expressions. As several dimensions exist, the primitive condition can be either of enumerative or parametric type, i.e. ECS, ECH, PCS and/or PCH. The condition expressions in the multidimensional case can be expressed by combining enumerative or parametric primitive conditions using the primitive logical operators, i.e. AND or OR.

The leaves of the tree presented in Fig. 5 and their combinations describe the possible condition statements that can occur. Exploring the different cases, we define the dominant and the non-dominant conditions.

Definition 6.2. The *dominant* conditions are the conditions that do not allow accesses in the inner dimension, if no accesses exist in the outer dimension.

For instance, the condition $(i \neq 5)\&\&(j > 8)$ does not allow the accesses of J pattern to occur when the I pattern has holes, i.e. the iterator i is equal to 5. However, the condition $(i \neq 5)||(j > 8)$ does allow accesses in J pattern, when the I pattern has holes. We cluster the different condition cases that can occur into five representative cases which have a common equation and we classify them as dominant or non-dominant.

(1) Dominant conditions:
   (a) The *case i* represents the primitive conditions ECS and ECH and their combination though the AND operator,
   (b) The *case ii* refers to the case of a PCH with $==$ comparison operator ($PCH_{=}$) and a $PCH_{=}$ combined with:
      i. the OR operator with $PCH_{=}$,
      ii. the AND operator with $PCH_{=}$, ECH, ECS.
   (c) The *case iii* describes the $PCH_{=}$ combined through the OR operator with ECS and ECH,

(2) Non-dominant conditions:

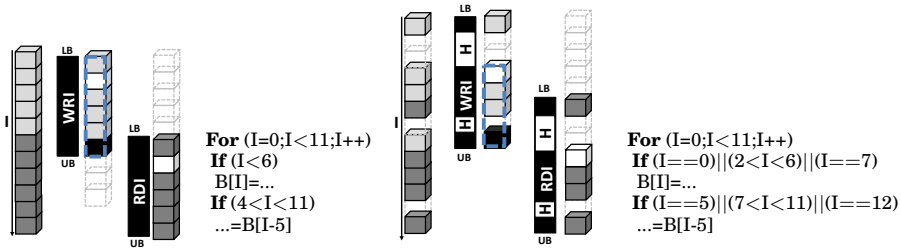


Fig. 6: Iteration spaces of WR and RD statements and corresponding codes for the Dominant segment with i) SIS (a)&(b) and ii) Non-Dominant segment with ISH (c)&(d).

(a) The *case iv* describes the combination of ECS and ECH through the OR operator.
(b) The *case v* describes a PCH with $\neq$ comparison operator ($PCH_{\neq}$) and a $PCH_{\neq}$ combined with:
   i. the AND operator with $PCH_{\neq}$, ECH, ECS,
   ii. the OR operator with PCH, ECH, ECS.

7. SEGMENT CASE: DOMINANT OR NOT-DOMINANT?

7.1. Dominant segment

The dominant segment is valid when the exploration window of the dimension i is smaller than or equal to a) the size of the dimension if the iteration space is SIS, or b) the length of the larger access segment if the iteration space is ISH: $(\mathsf{Diff_i} \leq \mathsf{IR_i})||(\mathsf{Diff_i} \leq max(\mathsf{SID_i}(k))$ for $k = 0, ..., \mathsf{N_i}$. The analysis of only the dominant segment can define the maximum number of alive elements for the dimension i. Hence, the size is given by the exploration window of dimension i, $\mathsf{PSize_0}$=$\mathsf{Diff_i}$.

Example 7.1. One example is given in Fig. 6(a) (with the corresponding code at Fig. 6(b)) for an iteration space without holes (SIS). The exploration window, i.e. the index difference between WR and RD, is 5, whereas the size of the SIS is 6 (this is the dominant segment marked by the blue dashed line). The first figure describes at the same time the behaviour of both the write (light grey boxes) and the read (dark grey boxes) statements. The next two figures describe the behaviour of each statement separately in order to understand how the accesses occur. The pattern of I dimension of WR statement is labelled as WRI and the pattern of RD statement is labelled as RDI. We will keep this notation in all the examples presented in the rest of the paper. The pattern of WR statement is $\{6A\}$, LB=-1, UB=6, IR=6, PS=6 and R=1 and the pattern of RD statement is $\{6A\}$, LB=4, UB=11, IR=6, PS=6 and R=1. The figure depicts one specific iteration, i.e. the WR is executed at the iteration 5 (black cell) and the next RD occurs at the iteration 6 (white cell). The element B[5] is written and the element B[1] is read in the next iteration. The number of alive elements is given by the exploration window, $\mathsf{Size_0}$=$\mathsf{PSize_0}$=$\mathsf{Diff_I}$=5.

7.2. Non-Dominant segment

In the non-dominant segment, all lengths of the access segments in dimension i are smaller than the exploration window. Hence, the size is $\mathsf{Size_{0-0}}$ = $\mathsf{Size_0}$ = $\mathsf{PSize_0}$ = $\mathsf{ADiff_i}$,

Pseudocode 3: Find maximum pattern section.

```
max=0; Part=0; Start=0
for (k=LB_i+1; k< UB_i+Diff_i; k++) do
    Size=0
    if (k==Start+SIR(part)) then
        Start+=SIR(part)
        Offset=0
    else Offset++
    Length=0
    while (Length+(SIR(Part)-Offset)≤ Diff_i) do
        if (PT(Part)==A) then Size+=SIR(Part)
        Length+=SIR(Part)-Offset
    Rem=Diff_i-Length+(SIR(Part)-Offset)
    if (ST(Part)==A) then Size+=Rem
    if (Size>max) then max=Size, k_max=k
```

where $\mathsf{ADiff_i}$ is given by the alive elements stored inside a section of the pattern which is as large as the exploration window $\mathsf{Diff_i}$.

The process to find the maximum number of alive elements and the pattern section is described by Pseudocode 3. The process searches the pattern in sections that have size $\mathsf{Diff_i}$. It starts from the first iterator until $\mathsf{UB_i} + \mathsf{Diff_i}$, since the pattern describes repetitive behaviour in the iteration space. In our algorithm, the parameter k gives the start of the pattern section to be searched. The parameter $Start$ is equal to the iterator values of the next pattern segment. The $Offset$ describes how far away is k from the start of the segment. If k is equal to the first iterator of a pattern segment, $Offset$ is 0. We add the segments with accesses as long as the total $Length$ is smaller than the exploration window $\mathsf{Diff_i}$. When the length goes over this value and the last pattern segment describes accesses, we add the required elements to reach a size equal to $\mathsf{Diff_i}$. Then, we increase k parameter and the process is repeated for the remaining sections of size $\mathsf{Diff_i}$. When several pattern sections exist which have the maximum number of elements, we explore the one that has an access just after the exploration window and it is placed in the middle of the pattern, as this is most resource consuming case.

However, the fact that the segment is not dominant may affect $\mathsf{Size_i}$ of Definition 1 used when we add a new dimension. This may occur in the case of the non-dominant conditions, because if an access occurs in the dimension i and a hole exists in 0 dimension, the element is still accessed. In this case, we need to also include these "hidden" elements. They are computed by finding the number of H that exist in the pattern section, $\mathsf{HDiff_0}$. We describe in Section 8.2 how they are used to compute $\mathsf{Size_i}$.

The equation to compute the terms $\mathsf{ADiff_i}$ and $\mathsf{HDiff_i}$ is given by Eq. 3 and 4, where $k_{\max}$ is the start of the pattern section with the maximum number of elements to be explored.

$$\mathsf{ADiff_i} = \sum_{l=k_{\max}}^{k_{\max}+\mathsf{Diff_i}} \mathsf{SID_i}(l) \tag{3}$$

$$\mathsf{HDiff_i} = \sum_{l=k_{\max}}^{k_{\max}+\mathsf{Diff_i}} \mathsf{HID_i}(l) \tag{4}$$

Example 7.2. The Fig. 6(c) (corresponding code in Fig. 6(d)) depicts a non-dominant segment in an iteration space with holes (ISH). The index difference is 5, whereas the maximum segment length of the access segments is 3. The pattern of RD statement is {1A 2H 3A 1H 1A} with LB=-1, UB=8, IR=8, PS=8 and R=1. The pattern section with length equal to the exploration window which has to be explored to find the maximum

Table III: Computation of the additional elements ASize_i.

Conditions	**Case Value**	**Conditions**	**Case Value**
$\mathbf{extra}_{\mathrm{PCH}_=}$		$\mathbf{extra}_{\mathrm{PCH}_{\neq}}$	
$\frac{\mathsf{Diff_i}}{\mathsf{PS}} < \mathsf{Diff_i}$	1	$\mathsf{Diff_i} > \mathsf{IR_i} - 1$	-1
$\mathbf{extra}_{\mathrm{PCH}_{=,\mathrm{A2A}}}$		$\mathbf{extra}_{\mathrm{WR}}$	
$\mathsf{CP}^{\mathrm{RD}} == \mathsf{H} \;\|\|$ $\mathsf{CP}^{\mathrm{RD}} == \mathsf{A} \;\&\&\; \mathsf{Diff_i} > 1$	1	$\mathsf{Diff_i} > \mathsf{LB_i}^{\mathrm{WR}} + 1$	$\sum_{l=0}^{\mathsf{N_i}} \mathsf{SID_i}(l)$
$\mathbf{extra}_{\mathrm{RD,A}}$		$\mathbf{extra}_{\mathrm{RD,H}}$	
$\mathsf{LB_i}^{\mathrm{RD}} + 1 < \mathsf{IR_i} \;\&\&$ $\mathsf{UB_i}^{\mathrm{RD}} > \mathsf{IR_i}$	$\mathsf{APos}(\mathsf{IR_i} - \mathsf{LB_i}^{\mathrm{RD}} - 1)$	$\mathsf{LB_i}^{\mathrm{RD}} + 1 < \mathsf{UB_i}^{\mathrm{WR}} \;\&\&$ $\mathsf{UB_i}^{\mathrm{RD}} > \mathsf{UB_i}^{\mathrm{WR}}$	$\mathsf{APos}(\mathsf{UB_i}^{\mathrm{WR}} - \mathsf{LB_i}^{\mathrm{RD}} - 1)$
$\mathsf{LB_i}^{\mathrm{RD}} + 1 \geq \mathsf{IR_i}$	$\sum_{l=0}^{\mathsf{N_i}} \mathsf{SID_i}(l)$	$\mathsf{LB_i}^{\mathrm{RD}} + 1 \geq \mathsf{UB_i}^{\mathrm{WR}}$	$\sum_{l=0}^{\mathsf{N_i}} \mathsf{SID_i}(l)$

number of alive elements is depicted by the blue dashed line. It starts at $\mathsf{SID}(2)$ and ends at $\mathsf{SID}(3)$. By adding the occurring accesses, Size_0=$\mathsf{ADiff_I}$=4, whereas the $\mathsf{HDiff_I}$ is 1.

8. CONDITIONS CASE: DOMINANT OR NOT-DOMINANT?

However, when the size is computed using more than one iteration in definition 1, the computation of the term $\mathsf{Size_i}$ is affected by the type of the conditions and the term $\mathsf{ASize_i}$ by the type of both the conditions and the outer dimension.

In case a condition is of parametric type, it couples the patterns of two dimensions and thus, they cannot be explored independently. However, the pattern of the outer dimension dominates. Hence, if the condition is of type:

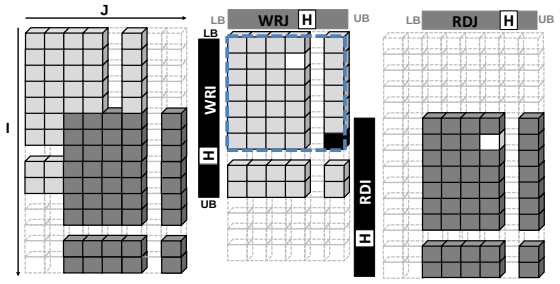
(1) $PCH_=$ (PCH with the $==$ comparison operator) or $PCH_{\&\&}$ (PCH with the $<\&\&>$ comparison operator), $\mathsf{Size_i}$ is 1. Due to the repetition of the parametric pattern, $\mathsf{ASize_i}$ is given $\mathrm{extra}_{\mathrm{PCH}_=}$ in Table III. When we have
   (a) one condition $PCH_=$, $\mathsf{ASize_i}$=$\mathrm{extra}_{\mathrm{PCH}_{=,\mathrm{A}}}$. The term $\frac{\mathsf{Diff_{out}}}{\mathsf{PS}}$ describes the repetition of the pattern in the outer exploration window. If the inner exploration window is larger than this repetition, an additional element must be stored.
   (b) more than one condition $PCH_=$, the term $\mathsf{ASize_i}$=$\mathrm{extra}_{\mathrm{PCH}_{=,\mathrm{A2A}}}$, where CP is the combined pattern of the initial $PCH_=$ patterns. These are the elements accessed in both PCH patterns, i.e. Access to Access (A2A), but not computed by the first equation of the *case ii*. For each of the A2A elements we verify if the combined pattern (CP) has a hole or not. If an H exists, this A2A element has not been read, so an extra element is stored. If an A exists, the extra element is stored only if the inner exploration window is larger than one. Otherwise, the A2A element has already been computed.

(2) $PCH_{\neq}$, the term $\mathsf{Size_i}$ is equal to $\mathsf{IR_i}$ for the partial size computed by $\mathsf{ADiff_i}$ and equal to $\mathsf{IR_i} - 1$ for the partial size computed by $\mathsf{HDiff_i}$.

8.1. Dominant conditions

The dominant conditions imply that an access in an element is valid when all dimensions are accessing this element. Hence:

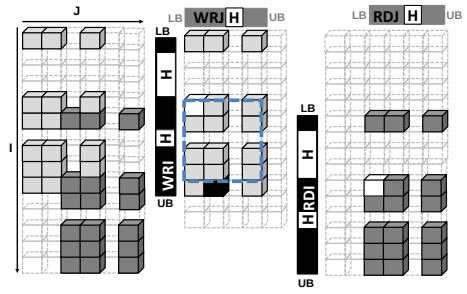
(1) $\mathsf{Size_i}$ is computed by adding only the elements that are accessed in the i dimension, $\sum_{l=0}^{\mathsf{N_i}} \mathsf{SID_i}(l)$.
(2) $\mathsf{ASize_i}$ is affected by the type of the outer dimension.



(a)

```
For (I=0;I<15;I++)
 For (J=0;J<8;J++)
  If (I<7)||(7<I<10) && (J<4)||(4<J<6)
   B[I][J]=...
  If (4<I<12)||(12<I<15) && (1<J<6)||(6<J<8)
   ...=B[I-5][J-2]
```

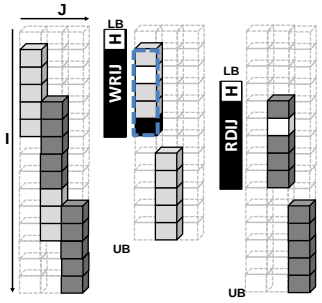
(b)



(c)

```
For (I=0;I<15;I++)
 For (J=0;J<6;J++)
  If ((I==0)||(I>3)&&(I<6)||(I>6)&&(I<10))&&((J<2)||(J==3))
   B[I][J]=...
  If ((I==5)||(I>8)&&(I<11)||(I>11))&&((J>1)&&(J<4)||(J ==5))
   ...=B[I-5][J-2]
```

(d)



(e)

```
For (I=0;I<15;I++)
 For (J=0;J<3;J++)
  If (I>6J)&&(I>6(J+1))&&(J<2)
   B[I][J]=...
  If (J>0)&&(I>6J-3)&&(I>6(J+1)-3)
   ...=B[I-3][J-1]
```

(f)

Fig. 7: Iteration spaces for WR and RD statements and corresponding codes for the Dominant conditions for Dominant Outer Dimension with dominant segments for 1) case i for ISH (a)&(b), 2) non-dominant segment for case i for ISH (c)&(d) and 3) case ii for ISH (e)&(f).

8.1.1. Dominant outer dimension. When the outer dimension is dominant, it means that the behavior of the pattern in the iteration after the exploration window is always A. This access means that the WR statement of the outer dimension is always executed. When the dominant segment is larger than the exploration window, it implies that the outer dimension is always dominant. This access means that potentially additional elements may occur in the i dimension. Hence, the term $\mathsf{ASize_i}$ is computed by adding the accessed segments in the pattern i for a length equal to the exploration window i, which is given by $\sum_{l=0}^{\mathsf{Diff_i}} \mathsf{SID_i}(l)$.

In the next paragraphs we provide several illustrative examples to show how the proposed methodology computes the size. In addition, we provide the Pseudocode 4 to illustrate in detail the computation process for the cases i and iv.

Pseudocode 4: Size computation for cases i and iv.

```
if ((condition type P1 == 'ECH'||'ECS') && (condition type P2 =='ECH'||'ECS')) then
    if (operation=='AND') then                                   /* Dominant condition */
        Size_j = sum_{l=0}^{N_j} SID_j(l)
        if (WR_iteration=='A') then                              /* Dominant dimension */
            ASize_j = sum_{l=0}^{Diff_j} SID_j(l)
        else                                                     /* Non-dominant dimension */
            ASize_j = 0
    if (operation=='OR') then                                    /* Non-dominant condition */
        Size_j = IR_j
        if (WR_iteration=='A') then                              /* Dominant dimension */
            ASize_j = Diff_j
        else if (WR_iteration=='H') then                         /* Non-dominant dimension */
            if (pattern_section==0) then                         /* First iteration */
                ASize_j = extra_WR
            else                                                 /* Middle iteration */
                ASize_j = max(extra_WR, extra_RD)
if (Dominant Segment) then
    Size_i, j = Diff_i * Size_j + ASize_j
else
    Size_i, j = ADiff_i * Size_j + HDiff_i * sum_{l=0}^{Diff_j} SID_j(l) + ASize_j
```

Example 8.1. The Fig. 7(a) (code in Fig. 7(b)) describes an example where the I and J patterns describe an ISH and the I pattern has dominant segment. The condition is dominant and belongs to *case i*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by $\sum_{l=0}^{\mathsf{N_j}} \mathsf{SID_j}(l)$, as we are in the dominant condition case, and ASize_j is given by $\sum_{l=0}^{\mathsf{Diff_j}} \mathsf{SID_j}(l)$, as the outer dimension is dominant. The final equation is given in the top-left part of Table IV. The dominant segment is marked by the blue dashed line. The pattern RDI is {7A 1H 2A}, LB=-1, UB=10, IR=10, PS=10 and R=1. The pattern RDJ is {4A 1H 1A}, LB=-1, UB=6, IR=6, PS=6 and R=1. The exploration window is 5 in I dimension and 2 in J dimension. The number of alive elements is given by 5*(4+1)+2=27, where 5 is $\mathsf{Diff_I}$, 4 is the length of the first accessed segment and 1 is the length of the second accessed segment in J pattern, and 2 is $\sum_{l=0}^{\mathsf{Diff_J}} \mathsf{SID_J}(l)$.

Example 8.2. In Fig. 7(c) describes an example where the I and J patterns describe an ISH and the I pattern has a non-dominant segment. The condition is dominant and belongs to *case i*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by $\sum_{l=0}^{\mathsf{N_j}} \mathsf{SID_j}(l)$, as we are in the dominant condition case, and ASize_j is given by $\sum_{l=0}^{\mathsf{Diff_j}} \mathsf{SID_j}(l)$, as the outer dimension is dominant. The final equation is given in the bottom-left part of Table IV. The pattern RDI is {1A 3H 2A 1H 3A}, LB=-1, UB=10, IR=10, PS=10 and R=1. The pattern RDJ is {2A 1H 1A}, LB=-1, UB=4, IR=4, PS=4 and

Table IV: Equations for several representative cases. The black terms describe the size due to I exploration window and the gray terms the additional elements due to J exploration window

Dominant segment		
Case	**Dominant outer dimension**	**Non-dominant outer dimension**
i	$\mathsf{Diff_I} * \sum_{l=0}^{\mathsf{N_J}} \mathsf{SID_J}(l) + \sum_{l=0}^{\mathsf{Diff_J}} \mathsf{SID_J}(l)$	$\mathsf{Diff_I} * \sum_{l=0}^{\mathsf{N_J}} \mathsf{SID_J}(l)$
ii	$\mathsf{Diff_I} * 1 + \mathrm{extra_{PCH_=}}$	$\mathsf{Diff_I} * 1$
iv	$\mathsf{Diff_I} * \mathsf{IR_J} + \mathsf{Diff_J}$	$\mathsf{Diff_I} * \mathsf{IR_J} + max(\mathrm{extra_{WR}}, \mathrm{extra_{RD,A}})$
Non-dominant segment		
Case	**Dominant outer dimension**	**Non-dominant outer dimension**
i	$\mathsf{ADiff_I} * \sum_{l=0}^{\mathsf{N_J}} \mathsf{SID_J}(l) + \sum_{l=0}^{\mathsf{Diff_J}} \mathsf{SID_J}(l)$	$\mathsf{ADiff_I} * \sum_{l=0}^{\mathsf{N_J}} \mathsf{SID_J}(l)$
ii	$\mathsf{ADiff_I} * 1 + \mathrm{extra_{PCH_=}}$	$\mathsf{ADiff_I} * 1$
iv	$\mathsf{ADiff_I} * \mathsf{IR_J} + \mathsf{HDiff_I} * \sum_{l=0}^{\mathsf{N_J}} \mathsf{SID_J}(l) + \mathsf{Diff_J}$	$\mathsf{ADiff_I} * \mathsf{IR_J} + \mathsf{HDiff_I} * \sum_{l=0}^{\mathsf{N_J}} \mathsf{SID_J}(l) + max(\mathrm{extra_{WR}}, \mathrm{extra_{RD}})$

R=1. The exploration window is 5 and 2 for I and J dimensions, respectively. The size is (2+2)*(2+1)+2=14, where (2+2) is the summation of the accesses in a pattern section of length 5, (2+1) is the summation for the accesses in dimension J and 2 is the summation of the accesses in the exploration window J.

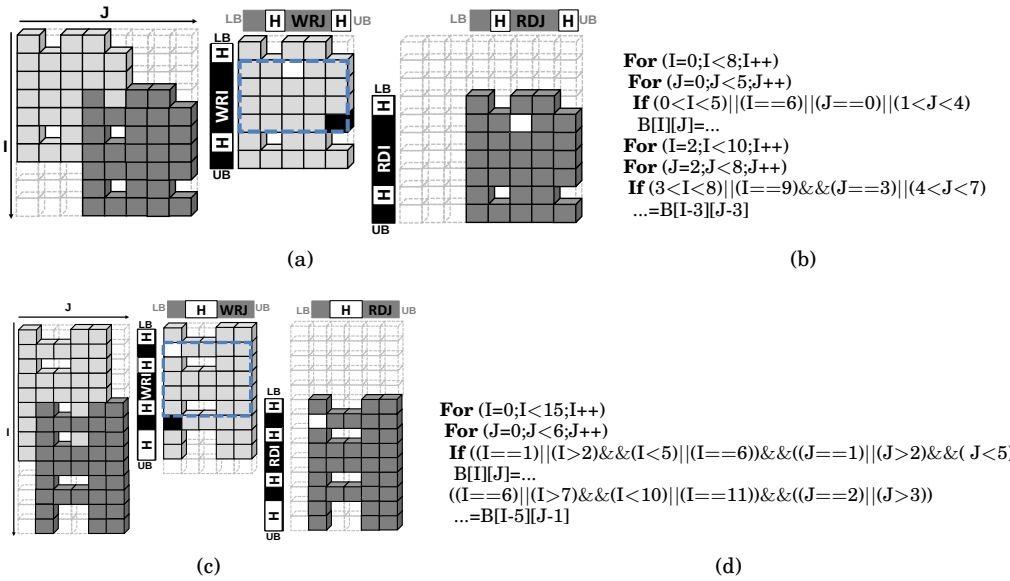
Example 8.3. Fig. 7(e) (code in Fig. 7(f)) describes an example where the I and the J are coupled through a PCH. The I pattern has dominant segment. The condition is dominant and belongs to *case ii*. The final equation is given in the top-left part of Table IV. The pattern RDI is {1H 5A}, LB=-1, UB=12, IR=12, PS=6 and R=2. The exploration window is 3 for I dimension and 1 for J dimension. The number of elements is 3*1+1=4, where 3 is $\mathsf{Diff_I}$, 1 is the size of J dimension due the PCH condition and 1 is the additional element that needs to be stored due to $\mathsf{Diff_J}$. We observe that when B[5] is written, B[1] has not yet been read and that still needs to be stored. If in this example the exploration window J was 0, the number of elements is 3, because when B[5] is written, B[1] has already been read.

8.1.2. Non-dominant outer dimension. In contrast to the previous case, the behavior of the pattern in the iteration after the exploration window is always H. Therefore, the term $\mathsf{ASize_i}$ equals to zero because at least in one dimension (dimension 0) the elements are not accessed. Several final equations for the already presented cases i and ii and for the non-dominant outer dimension are given in the top-right part of Table IV.

8.2. Non-dominant conditions

In the case of the non-dominant conditions, at least an access in one pattern of a dimension is enough to access the element.

(1) the term $\mathsf{Size_i}$ depends on the type of the segment:
   (a) Dominant segment: all elements are accesses in the 0 dimension and thus the size is given by the complete size of i dimension, $\mathsf{IR_i}$.
   (b) Non-Dominant segment: even if a hole exists in 0 dimension, the element is still accessed due to the access occurring in the i dimension. These "hidden" elements are given by $\mathsf{HDiff_0}$. Hence, the size derives by multiplying $\mathsf{ADiff_0}$ with




```
For (I=0;I<8;I++)
 For (J=0;J<5;J++)
  If (0<I<5)||(I==6)||(J==0)||(1<J<4)
   B[I][J]=...
For (I=2;I<10;I++)
For (J=2;J<8;J++)
 If (3<I<8)||(I==9)&&(J==3)||(4<J<7)
  ...=B[I-3][J-3]
```

```
For (I=0;I<15;I++)
 For (J=0;J<6;J++)
  If ((I==1)||(I>2)&&(I<5)||(I==6))&&((J==1)||(J>2)&&( J<5))
   B[I][J]=...
  ((I==6)||(I>7)&&(I<10)||(I==11))&&((J==2)||(J>3))
   ...=B[I-5][J-1]
```

Fig. 8: Iteration spaces for WR and RD statements and corresponding codes for Non-Dominant conditions for Dominant Outer Dimension for the case iv with 1) dominant (a)&(b) and 2) non-dominant (c)&(d) segments.

$\mathsf{IR_i}$ (as in the dominant segment case) and multiplying HDiff_0 by the accessed elements in the dimension i, $\sum_{l=0}^{\mathsf{N_i}} \mathsf{SID_i}(l)$.

(2) $\mathsf{ASize_i}$ derives from the type of the outer dimension and the position of the segment.

8.2.1. Dominant outer dimension. In this case $\mathsf{ASize_i}$ is computed by the i exploration window, $\mathsf{Diff_i}$.

Example 8.4. The Fig. 8(a) (code in Fig. 8(b)) describes an example where the pattern I and the pattern J describe an ISH and I pattern has a dominant segment. The condition is non-dominant and belongs to *case iv*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by IR_j, as we are in the non-dominant condition case, and ASize_j is given by $\mathsf{Diff_j}$ as the outer dimension is dominant. The final equation is given in the top-left part of Table IV. The pattern RDI is {1H 4A 1H 1A}, LB=-1, UB=7, IR=7, PS=7 and R=1. The pattern RDJ is {1A 1H 2A 1H}, LB=-1, UB=5, IR=5, PS=5 and R=1. The exploration window is 3 in I dimension and 3 in J dimension. The dominant segment is marked by the dashed blue line. The number of alive elements is 3*5+3=18, where 3 is I exploration window, 5 is the size of J loop, and 3 is J exploration window.

Example 8.5. In Fig. 8(c) describes an example where the pattern I and the pattern J describe an ISH and I pattern has a non-dominant segment. The condition is non-dominant and belongs to *case iv*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by IR_j, as we are in the non-dominant condition case, and ASize_j is given by $\mathsf{Diff_j}$ as the outer dimension is dominant. The final equation is given in the bottom-left part of Table IV. The pattern RDI is {1H 1A 1H 2A 1H 1A 2H}, LB=-1, UB=9, IR=9, PS=9 and R=1. The pattern RDJ is {1A 2H 2A}, LB=-1, UB=5, IR=5, PS=5 and R=1. The exploration window is 5 in I dimension and 1 in J dimension. The

pattern section with the maximum number of alive elements in I dimension is {1A 1H 2A 1H}. The final number of elements is (1+2)*5+(1+1)*(1+2)+1 = 22, where (1+2) is the summation of the accesses in I dimension, 5 is the size of J dimension, (1+1) is the summation of the holes in the I dimension, (1+2) is the summation of the accesses in J dimension and 1 is J exploration window.

8.2.2. Non-dominant dimension. The term $\mathsf{ASize_i}$ depends on the position of the dominant segment (or the pattern section with the maximum number of accessed elements in the case of the non-dominant segment):

(1) First iteration: In this case no elements have been stored before the execution of the dominant segment. Hence, the additional elements derive only from the writes after the dominant segment. The term $\mathsf{ASize_i}$ is given by $\mathrm{extra}_{\mathsf{WR}}$ computed in Table III. If the exploration window i is smaller than the first written element in i dimension, i.e. $\mathsf{LB_i^{WR}} + 1$, no elements have been written. Otherwise, the number of additional elements is given by finding the accessed elements inside the exploration window i.
(2) Middle iteration: The additional elements in the dimension i are now given by the maximum between the elements that have been written just after the dominant segment and the elements that have been written, but and not yet read, just before the dominant segment, i.e. $max(\mathrm{extra_{WR}}, \mathrm{extra_{RD}})$. The terms $\mathrm{extra_{WR}}$ and $\mathrm{extra_{RD}}$ are computed in Table III. The term $\mathrm{extra_{WR}}$ is the same with the case where the dominant segment is placed in the first iteration. The term $\mathrm{extra_{RD}}$ depends on the behaviour of the pattern in the outer dimension before the dominant segment. If in the outer dimension the pattern has an:
   (a) Access: If the last occurred read is placed lower than or equal to the size of the loop i, $\mathsf{IR_i}$, the elements written before the dominant segment have already been read. Hence, the term $\mathrm{extra_{RD,A}}$ is 0. Otherwise, the additional elements are computed by adding the accesses occurring from the position $\mathsf{IR_i} - \mathsf{LB_j^{RD}} - 1$ until the end of i loop.
   (b) Hole: The term $\mathrm{extra_{RD,H}}$ is 0, when all the elements before the exploration window have been read. Otherwise, the additional elements are computed by adding the accesses from the position $\mathsf{UB_i} - \mathsf{LB_i^{RD}} - 1$ up to the end of i loop.

Example 8.6. Fig. 9(a) shows an example where the I and J patterns describe an ISH and the I pattern has a dominant segment which is placed in the first iteration. The condition is non-dominant and belongs to *case iv*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by IR_j, as we are in the non-dominant condition case, and ASize_j is given by $max(extra_{\mathrm{WR}}, extra_{\mathrm{WR}})$ as the outer dimension is non-dominant. The final equation is given in the top-right part of Table IV. The pattern RDI is {4A 2H}, LB=-1, UB=6, IR=6, PS=6 and R=1, the pattern RDJ is {1A 1H 2A}, LB=-1, UB=4, IR=4, PS=4 and R=1. The exploration window is 4 and 2 in I and J dimension, respectively. The size is 4*5+1=21, where 4 is I exploration window, 5 is the size of J dimension and 1 is the additional elements required to be stored in J dimension, since one access exist in J exploration window.

Example 8.7. Fig. 9(b) shows an example where the patterns I and J describe an ISH and the I pattern has a dominant segment which is placed in a middle iteration. The condition is non-dominant and belongs to *case iv*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by IR_j, as we are in the non-dominant condition case, and ASize_j is given by $max(extra_{\mathrm{WR}}, extra_{\mathrm{WR}})$ as the outer dimension is non-dominant. The final equation is given in the top-right part of Table IV. The pattern RDI is {2H 4A 2H}, LB=-1, UB=8, IR=8, PS=8 and R=1. The pattern RDJ is {1A 1H 2A}, LB=-1, UB=4, IR=4, PS=4 and R=1. The index difference is 4 and 1 in I and J dimension, respectively. The size is 4*5+1=21, where 4 is I exploration window, 5 is the

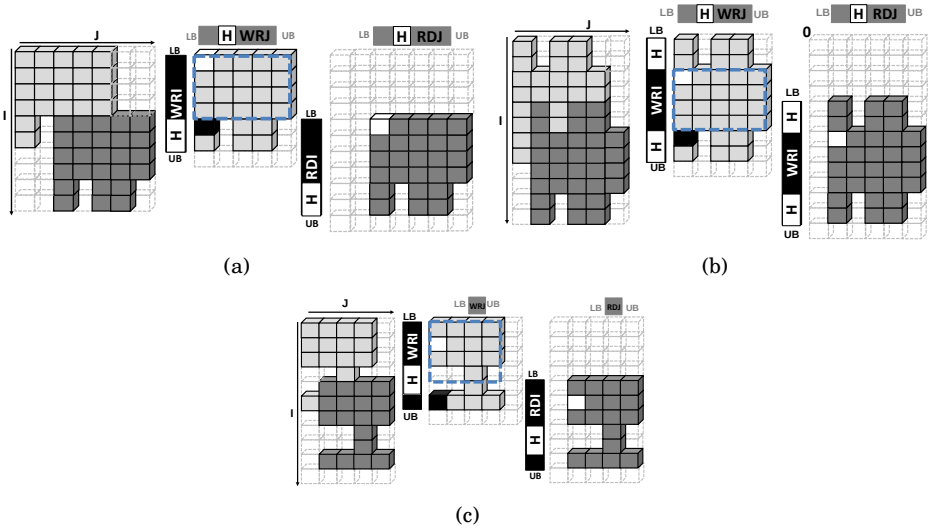


Fig. 9: Iteration spaces for the Non-dominant conditions for case iv with i) a Dominant Segment and Non-Dominant Outer Dimension placed 1st iteration (a) and in a middle iteration (b), and ii) a Non-Dominant Segment with Non-Dominant Outer Dimension placed 1st iteration (c).

size of J dimension and 1 is the extra accessed element for an exploration window of length 1 in J dimension ($\text{extra}_{\text{WR}} = 1$ and $\text{extra}_{\text{RD,A}} = 0$).

Example 8.8. Fig. 9(c) shows an example where the patterns I and J describe an ISH, the I pattern has non-dominant segment which is placed in the 1st iteration. The condition is non-dominant and belongs to *case iv*. The computation of the size is depicted in Pseudocode 4, where Size_j is given by IR_j, as we are in the non-dominant condition case, and ASize_j is given by $max(extra_{\text{WR}}, extra_{\text{WR}})$ as the outer dimension is non-dominant. The final equation is given in the bottom-right part of Table IV. The pattern RDI is $\{$3A 2H 1H$\}$, LB=-1, UB=6, IR=6, PS=6 and R=1. The pattern RDJ is $\{$1A$\}$, LB=1, UB=3, IR=1, PS=1 and R=1. The index difference is 4 and 1 for I and J dimensions. The number of elements is 3*5+1*1+0=16.

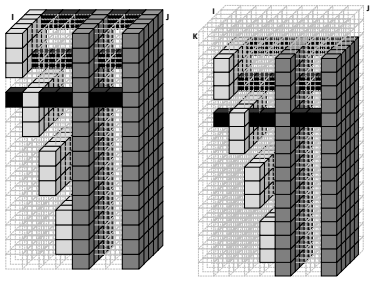
9. EVALUATION

In section 9.1, we describe how our approach is applied in a complex example, whereas section 9.2 presents the experimental results.

9.1. Demonstration case study

In this section we apply the proposed methodology for the example in Fig. 10(a). The analysis step provides the information: three for nested loops, loop order I-J-K, a PCH condition which couples iterator I and J (PCH1) of $< \&\& >$ type, two ECH conditions on iterator J (ECH1 and ECH2) and a PCH condition which couples iterator I and K (PCH2). All conditions are combined with || operation. The exploration window I is 2, the exploration window J is 1 and the exploration window K is 1. The translation step creates the patterns of the PCH conditions per pair of dimensions: for condition $(I \geq 4J)\&\&(I \leq 4J + 2)$, LB of loop I is -1, UB=1024, PS is 4, IR is 1024, R is 256 and the PCH1 pattern is $\{$3A 1H$\}$. The light grey elements of Fig. 10(b) depict the first part of the accessed elements of PCH1 of the three dimensional array. The ECH1 condition

```
For (I=0; I<1024; I++)
 For (J=0; J<256; J++)
  For (K=0; K<512; K++)
   If ((I≥4J)&&(I≤4J+3))||
    (J==7)||(J==4)||(I==2K)
     B[I][J][K]=...
For (I=2; I<1026; I++)
 For (J=1; J<257; J++)
  For (K=1; K<513; K++)
   If ((I≥4J)&&(I≤4J+3))||
    (J==8)||(J==5)||(I==2K)
     B[I-2][J-1][K-1]=...
```



(a) (b) (c)

Fig. 10: Demonstration case: (a) Code, (b) initial part of the iteration space for the WR iterations and the accessed elements. Each colour corresponds to a condition and (c) the execution of the RD statements in the iteration space.

of $J == 4$ creates a ECH2 of $\{$1A$\}$ pattern with LB=3, UB=5, PS is 1, IR is 1, R is 1. The ECH2 condition of $J == 7$ creates a ECH1 of $\{$1A$\}$ pattern with LB=6, UB=8, PS is 1, IR is 1, R is 1. The ECH1 and ECH2 access the dark grey elements in Fig. 10(b). The PCH2 condition $I == 2K$ has LB=-1 UB=1024, PS is 2, IR is 1024, R is 512 and the PCH2 pattern is $\{$1A 1H$\}$. The PCH2 access the black elements of Fig. 10(b). In the computation step, the process of computing the final storage size starts from the outer dimension. The combined pattern of ECH1 and ECH2 is $\{$1A 2H 1A$\}$ pattern with LB=3, UB=8, PS is 5, IR is 5, R is 1. The dominant segment case in outer dimension is selected since ECH is in inner dimensions and due to || operation the complete I dimension is accessed. The size is computed as $Size_{I,J}$=2*(1+1)+1+1*2+1=8. Then, we proceed to the next dimension K. The propagation multiplies the partial storage size $Size_{I,J}$ by size of K due to || operation. For the K index difference, the dominant segment in outer dimension and the non-dominant outer dimension case is valid. The partial storage size of K is given by *case ii* in Table IV top-right part. The additional elements in all dimensions due to K index difference are pruned. Hence, the additional elements in K dimension are given by the size in K dimension multiplied by IR_J minus the summation of A in the combined pattern of I and J dimensions. The final result is $Size_{I,J,K}$=8*512+1*(256-3)=4,349.

9.2. Experimental Results

We compared the exploration time of our method with the enumerative and the polyhedral approaches, which provide optimal results, and the approximation approach, with overestimated results, with respect to the exploration time and to the maximum number of concurrently alive elements. The enumerative approach computes this maximum number by adding the number of stored elements between the write and the last read of an array element. The process is applied for all the elements and the maximum result defines the required size. The enumerative approach produces optimal results because all the cases are exhaustively explored and its exploration time provides an upper bound in the exploration time of the methodologies. When the symbolic/polyhedral approaches are applied in iteration spaces with holes, the number of disjoint accessed regions in the iteration space is increased due to the existence of holes. Hence, the number of polytopes and the number of linear equations which is required to describe the space is increased. In order to reduce the exploration time, the holes are considered as accesses decreasing the number of the required equations. To compute the exploration times for our experiments we have used the barvinok tool,

Table V: Dominant Segment in Outer Dimension

Alg.: Array	Loop bounds	Near-optimal				Approx.
		Size	Execution Time(ms)			Size
		(Elem.)	Proposed	Polytopes	Enumerative	(Elem.)
Atax*: A	(32)(32)	32	0.101	103	14	32
	(128)(128)	128	0.103	101	816	128
	(512)(512)	512	0.104	105	52,059	512
	(2,048)(2,048)	2,048	0.101	100	-	2,048
Reg_detect*: path	(20)(20)	21	0.104	98	4	21
	(60)(60)	61	0.102	97	89	61
	(200)(200)	201	0.104	98	3,070	201
	(600)(600)	601	0.102	97	86,142	601
	(2,000)(2,000)	2001	0.100	95	-	2001
Gsm**: Update_residual_signal P3: drp	(100)(3)(40)	80	0.100	87	153	80
	(100)(3)(80)	160	0.102	89	609	160
	(100)(3)(120)	240	0.104	85	1,341	240
	(100)(3)(240)	480	0.103	81	5,377	480
	(100)(3)(460)	920	0.104	87	19,166	920
Jacobi-2D*: A	(128)(32)	34	0.177	106	34	34
	(1)(128)(64)	66	0.181	104	113	66
	(1)(128)(1,000)	1,002	0.190	104	23,440	1,002
	(1)(128)(8,000)	8,002	0.175	103	1,577,880	8,002
Doitgen*: sum	(10)(10)(10)	111	0.108	331	42	111
	(16)(16)(16)	273	0.104	329	452	273
	(28)(28)(28)	813	0.103	330	6,586	813
	(32)(32)(32)	1,057	0.103	331	12,954	1,057

-:Memory Error produced during simulation.
(*) Polybench, (**) MediaBench, (-) Memory Overflow

barvinok-0.36 [Verdoolaege et al. 2013]. The approximation approach estimates the shape by considering the holes in the iteration spaces between the WR and the RD of the elements as accesses. In this way, we obtain a less pessimistic approximation compared to other approaches, which tend to also simplify the outline of the shape, such as approximated convex hulls or approximated enumerators, increasing the over-approximations. For instance, in Fig. **??** the approximation makes the holes of $2x2$ and $1x2$ to accesses to have a convex hull $11x5$. We compare the result derived from an efficient approximation applied in irregular iteration spaces, which computes the number of elements as if the iteration space was solid but only for the exploration window. Otherwise the approximation is quite pessimistic, e.g. the heuristic in [Ramanujam et al. 2001] applied when the symbolic techniques cannot be applied due to access irregularity. The approximation is computed by considering the holes in the exploration windows as accesses and then the size can be computed based on the equations for the dominant segment of Section IV. In this way, an optimistic upper bound is achieved.

We present the exploration time and the storage size (in number of required elements) for several benchmarks from the PolyBench [Pouchet et al. 2012] and the MediaBench [Lee et al. 1997]. For each benchmark, different sizes in the number of accesses in the overall iteration space are tested by increasing the loop bounds to explore the scalability of the approaches. The benchmarks have been explored to identify arrays with overlapping writes and reads. We have manually extracted the access schemes for the condition and access statements and we have provided the basic corresponding patterns from the codes as input to our implementation. Therefore, the time provided in the experiments coves the translation step and the computation step. The presented benchmarks are selected to cover the different computation cases presented in this paper. Table VII describes the patterns of each benchmark and in which case

Table VI: Non-dominant Segment in Outer Dimension

Alg.: Array (init.bounds)	Loop Bounds	Near-optimal				Approx.
		Size (Elem.)	Execution Time(ms)			Size (Elem.)
			Proposed	Polytopes	Enumerative	
Jpeg**: idct_2x2: wsptr	(128)(16)	84	0.901	514	132	132
	(256)(32)	164	0.896	517	2,558	260
	(512)(128)	644	0.902	511	44,244	1,028
	(1,024)(384)	1,924	0.899	513	5,842,418	3,076
Jpeg**: decompress_ smooth_data: coef_bits	(4)(228)	5	0.158	128	19	6
	(4)(468)	10	0.175	134	65	12
	(4)(708)	15	0.183	132	138	18
	(4)(948)	20	0.219	129	234	24
	(4)(1,188)	25	0.219	134	365	30
Jpeg**: idct_2x2: wsptr (un)	(128)(16)	80	0.798	516	113	132
	(256)(32)	160	0.793	512	2,543	260
	(512)(128)	640	0.795	517	44,143	1,028
	(1,024)(384)	1,920	0.792	505	5,843,374	3,076
Gauss-Seidel: A'	(32)(32)	31	0.161	275	13,701	34
	(500)(500)	499	0.162	273	47,432	502
	(1,000)(1,000)	999	0.164	265	397,212	1,002
	(2,000)(2,000)	1,999	0.163	270	3,066,945	2,002

(**) MediaBench

they belong in our methodology. We have selected to present the two dimensions case as it is more representative of realistic application codes. The time and size for the one dimension cases have also been verified, e.g. for y array in Atax benchmark the time of the proposed methodology is stable around 0.093 ms and the enumerative is from 1.486 - 86,270.273 ms depending on the number of accesses. We also provide one case study of a three dimensional case, i.e. the Doitgen benchmark in Table V.

Table VII: Characterization of the experiments.

Algorithm	#Array Dimensions	# Segments /pattern	Diff	Computation Case	
				Dominant	Non-dominant
Atax	2	i: 1 j: 1	i: 0 j: 1	Seg.,Cond.,Dim.	
Reg_detect	2	i: 1 j: 1	i: 1 j: 1	Seg., Dim.	Cond.
Gsm	2	i: 1 j: 1	i: 2*size of j j: 0	Seg., Cond.	Dim.
Jacobi-2D	2	i: 1 j: 1	i: 1 j: 1	Seg.	Dim., Cond.
Doitgen	3	i: 1 j: 1 k: 1	i: 1 j: 1 k: 1	Seg., Cond., Dim.	
Jpeg (wsptr)	2	i: 7 (A: 5, H: 3) j: 1	i: 8 j: 4	Cond., Dim.	Seg.
Jpeg (coef_bit)	2	i: 1 j: 2 (A: 5, H: 1)	i: 0-4 j: 6	Seg., Dim.	Cond.
Jpeg (wsptr-un)	2	i: 7 (A: 5, H: 3) j: 1	i: 8 j: 4	Cond.	Seg., Dim.
Gauss-Seidel	2	i: 3 (1H Size-2 A 1H) j: 3 (1H Size-2 A 1H)	i: 1 j: 1	Seg., Cond., Dim	

From the results provided for the dominant segment, e.g. benchmarks in Table V, they are similar to the results when we are applying the methods in the solid iteration spaces, i.e without holes, because the exploration window is smaller than the dominant segment. When the segment which defines the size is solid, the approximation is similar to the proposed methodology, which achieves optimal size with low time. For

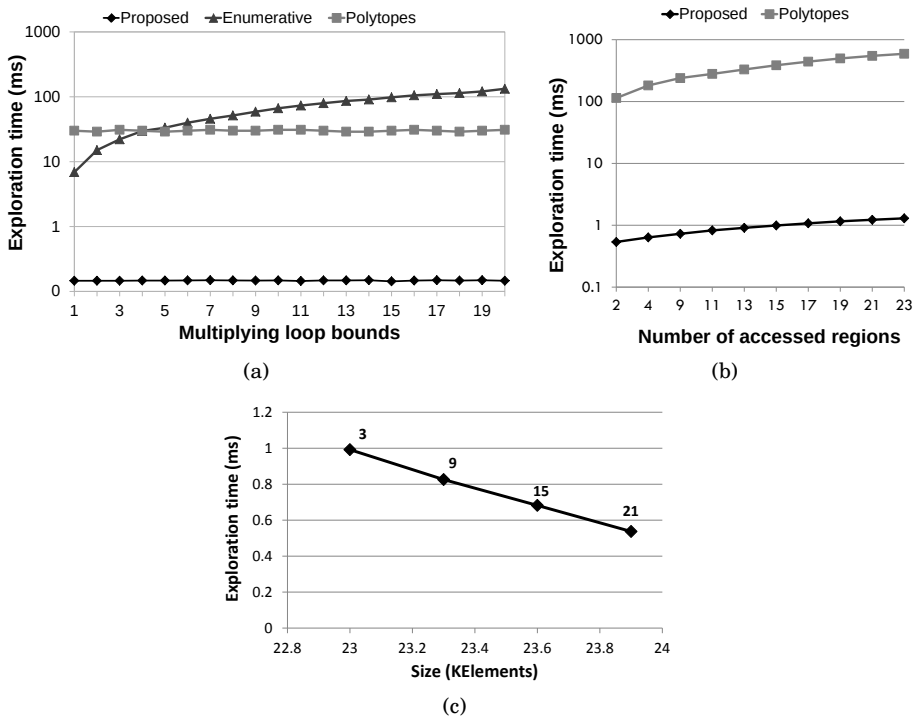


Fig. 11: Exploration times when the number of accesses is increased due to an increase in the (a) loop bounds, (b) number of accessed regions and (c) trade-offs between exploration time and optimal results.

the non-dominant segment case, e.g. benchmarks in Table VI, we searched the benchmarks for instantiation of the codes where holes are included in the exploration window. When holes exist in the iteration space, the proposed methodology also maintains optimal results and low exploration times. Due to the existing holes, the approximation leads to size overestimation as it finds an upper bound by considering the holes as accesses. We observe from the results of Table V an increase in the computed size for the Jpeg, and more precisely an over-approximation of 57.14% for the idct_2x2: wsptr, 20% for the coef_bit and 65% for the idct_2x2: wsptr (un) compared with the optimal case.

The size overestimation is increased with the increase in the loop bounds, when more than one loop dimensions exist. This occurs because when the approximation approach considers an H as an A in I dimension, the complete J dimension is considered as accessed. For instance, in the Jpeg benchmark for the wsptr cases in Table VI, although the cases are different, the approximation approach leads to the same results, as it cannot distinguish the two cases. The size loss in the non-dominant outer dimension case is larger, since the H of the iteration after the exploration window is considered as an A.

From the experimental results, the exploration time of the enumerative approach highly increases with the increase of the accesses in the iteration space. In contrast, the proposed methodology achieves optimal storage size with stable exploration time both for the SIS and the ISH spaces. The increase in the loop bounds modifies only the pattern parameters, which does not affect the methodology exploration time. The ex-

ploration time of our approach is dominated by the setup times to select the equations and the computation of their main primitive components, which is not affected by these parameters. A similar behaviour holds also for the exploration time of the polytope approach [Verdoolaege et al. 2013] when the loop bounds are increased, as depicted in Fig. 11(a), where we observe a gain of two orders of magnitude of our approach.

We observe that the exploration times of our method are quite close for almost all the benchmarks. The time is slightly increased only for the non-dominant segment cases due to the increase in the exploration window. This behaviour happens because we apply an exploration of the pattern of the outer dimension to find the pattern section with the maximum accesses. Therefore, we further explore the exploration time of the polytope approaches and the proposed method when the irregularity in the application array accesses is increased. Then, the number of accesses regions are increased, which is equivalent in increasing the number of polytopes required to describe the exploration space and the number of segments in the patterns of the proposed methodology. Fig. 11(b) depicts the exploration times using the barvinok and the proposed approach for different number of accessed regions. This is simulated in our benchmark by increasing the access statements in the inner kernel by unrolling to observe the timing behaviours. We observe a linear increase for the proposed methodology. From the obtained results, an increase of 84,7% in time occurs for the proposed methodology when the number of pattern segments is increased by a factor of 7, whereas the polytopes have an increase of 287.7%, which is larger by a factor of 3.4 compared to our method.

In addition, the proposed methodology provides a control mechanism for the exploration time which can reduce the number of pattern segments by considering a few small holes of a pattern as accesses and, thus, merging the pattern segments. Hence, our approach is capable of approximating the space in a controlled way in order to reduce the exploration time. Fig. 11(c) describes the size approximation and the decrease in the exploration time, when a pattern with a high number of segments occurs in the outer dimension. The optimal solution is 230*100, where 100 in the inner loop size and the value 230 derives by exploring 21 segments of the initial pattern {32A 1H 16A 1H 32A 1H 16A 1H 16A 1H 8A 1H 22A 1H 8A 1H 32A 1H 16A 1H 32A}. We assume an exploration window of 240, which is the size of the pattern. Then, we are approximating the space by considering each time 3 holes of size 1 as accesses and we thus obtain new patterns of 15, 9, and 3 segments to be explored. We observe a gain of 84,7% in decreasing the exploration time while we increase the calculated size by 4%. The loss in size and the gain in time due the approximation depends on the size of the holes and the accesses.

10. RELATED WORK

The existing techniques to compute the maximum number of concurrently alive elements can be enumerative, symbolic/polyhedral or approximations.

10.1. Enumerative approaches

The enumerative techniques describe each access to the array individually. An enumerative approach is used in the symbolic evaluation for background memory size estimation based on enumeration of the indexed signals of each index expressions [Nachtergaele et al. 1992]. Ref. [So et al. 2004] proposes a custom memory data layout, i.e. placement of the elements into the memory resources, focusing on parallelization of memory accesses based on the access pattern. Each access in the array access pattern is analyzed and the accesses of independent array elements are placed in separate partitions. The enumeration of the memory accesses is described through reference lists-based schemes without applying any reference compression [Paek et al. 2002]. The enumeration of memory accesses is usually achieved through profiling and instru-

mentation tools. A profiling based strategy to generate a memory access trace and a heuristic approach to exploit the scratchpad memory hierarchy is proposed in [Cho et al. 2007]. Ref [Ball et al. 1996] enumerates through profiling the iterations, where the memory is accessed, to derive the information for selecting candidates for data remapping [Palem et al. 2002]. Ref. [Rubin et al. 2002] searches all possible memory data layouts by iteratively prototyping candidate data layouts and evaluating them on a representative trace of memory accesses. SHMAP [Dongarra et al. 1990] tool annotates the application and collects memory reference traces for arrays. Gleipnir [Janjusic et al. 2011] collects memory access traces and associates each access with the corresponding internal structure. Pin [Luk et al. 2005] provides an instrumentation to create a trace of address and size of memory instructions. In [Weidendorfer et al. 2004] the data accesses are profiled through the instrumentation framework presented in [Nethercote et al. 2006].

The enumerative approaches are optimal, but they lack scalability. As the application size is increased, the number of accesses is increased highly affecting the exploration time. In contrast, the exploration time of the proposed approach remains stable when the number of memory accesses is increased, as depicted from the experimental results in the evaluation section.

10.2. Symbolic/polyhedral approaches and approximations

Other techniques are symbolic and apply solvers to compute the size. For instance, in linear constraint-based schemes the array accesses are expressed as convex regions in a geometrical space [Paek et al. 2002]. The storage requirements derive from the integer points inside the convex regions of accesses. The symbolic approaches are scalable and near-optimal in solid iteration spaces, as they efficiently represent the edges of one convex region. For instance, simplified constraint-based forms (e.g. [Balasundaram and Kennedy 1989]) are used to describe solid iteration spaces, e.g. diagonal or triangular shapes, which are not applicable in iteration spaces with holes. Other symbolic approaches with extensions can efficiently handle piece-wise regular spaces. For instance, triple notation [Shen et al. 1990], i.e. lower bound, upper bound and stride per dimension, has been used to describe regular spaces. Vectors have also been explored for storage size management. Ref. [Clauss et al. 2000] focuses on spatial locality optimization using utilization vectors to describe array references. Ref. [Jang et al. 2011] uses memory access vectors, the loop nest depth and the array dimension. In [Kandemir 2001] an access matrix describes through the loop nest the array accesses to explore data locality. Distance vectors with data access matrices are used, which are applicable for uniform references [Ramanujam et al. 2001]. The survey in [Panda et al. 2001] describes several symbolic techniques for estimation of storage requirements. Polytope theory is commonly used for regular spaces, e.g. the iteration space is represented by placing polytopes of signals in a common place with ILP techniques [Kjeldsberg et al. 2003]. Mature tools are supporting the polytope theory, such as [Verdoolaege et al. 2013]. The work presented in [Clauss et al. 2012] and in [Verdoolaege et al. 2007] describes the polyhedral approach which computes exact results by viewing an instance, or iteration, of each statement as an integer point in a polyhedron. Estimation on storage requirements with a partial fixed ordering through polytopes is proposed in [Kjeldsberg et al. 2004]. IMEC Atomium [Catthoor et al. 1998] supports memory related steps through interactive or in a more automated way based on the polyhedral dependency graph [van Swaaij et al. 1992]. The Data Transfer and Storage Exploration (DTSE) methodology uses the polyhedral dependency graph to explore the memory data layout optimization step. In [Wuytack et al. 1998] a data access graph based on polytopes is used to describe all the memory operations in time for a given array, which is used as input to the data reuse exploration and decision step of the memory hier-

archy design. Ref [Cong et al. 2011] applies polytope theory to memory partition and scheduling problem. Philips Phideo [Lippens et al. 1993] is mainly oriented to stream-based video applications and represents the iteration space as linear function of the iteration index. In [Darte et al. 2005] an efficient representation using lattice matrices is used to solve the memory allocation problem for iteration spaces with regular holes. Ref. [Seghir et al. 2012] proposes a lattice intersection approach to count the integer points in $\mathcal{Z}$-polytopes. Ref. [Clauss et al. 2009] describes an approach to maximize a parametrized and multivariate polynomial defined over a parametrized convex domain using unions of linear equations, which provides an upper bound to the required resources. SUIF [Maydan et al. 1993] and PIPS [Creusillet and Irigoin 1996] add additional constraint to the linear constraint-based representation to deal with a hole. Additional symbolic constraints are needed to describe the different regions increasing the exploration time [Paek et al. 2002]. To reduce the exploration time, the access regions have to be widen in order to form a convex hull which approximates the size. A wide survey of the existing methods and the relevant results for the approximations of convex sets using polytopes is provided in [Bronstein 2008; Barany 2007]. In [Meister and Verdoolaege 2008] a set of techniques which compute polynomial approximations of the quasi-polynomials are presented. Other techniques [Ramanujam et al. 2001] approximates the number of distinct references of non-uniform access statements based on the values of the index expressions on the loop bounds. Techniques to compute the approximated convex hulls are presented in [Bentley et al. 1982].

The applications with design-time unknown array accesses, e.g. due to data dependent conditions, create several codes instances with high irregularity, which are described by irregularly placed holes and potentially a high number of irregularly placed regions in the geometrical space. These geometrical shapes are not polytopes or lattices, and therefore existing approaches require complex models to describe them. Hence, the exploration time of the solvers applied in these complex formulations is increased with an increase over the irregularities. The approximations can simplify the shapes but without control over the resources overestimation. The contribution of the proposed methodology is to provide optimal results for irregular iteration spaces and a mechanism which controls the potential approximations applied, as depicted by the experimental results. However, compared with the matured polytope approaches the proposed method in its current first form does not deal with parametric expressions of the application variables.

11. CONCLUSIONS

We have presented a methodology to compute in a scalable and near-optimal way the maximum number of concurrently alive elements in iteration spaces with irregular holes and overlapping write and read accesses. We propose a set of computation cases with closed form equations to calculate the size. Based on the results conducted, the proposed methodology achieves near-optimal size with low exploration time.

REFERENCES

BALASA, F., KJELDSBERG, P. G., VANDECAPPELLE, A., PALKOVIC, M., HU, Q., ZHU, H., AND CATTHOOR, F. 2008. Storage estimation and design space exploration methodologies for the memory management of signal processing applications. *J. Signal Process. Syst. 53,* 1-2, 51–71.

BALASUNDARAM, V. AND KENNEDY, K. 1989. A technique for summarizing data access and its use in parallelism enhancing transformations. *SIGPLAN Not. 24,* 7, 41–53.

BALL, T. ET AL. 1996. Efficient path profiling. In *MICRO*. IEEE, USA, 46–57.

BARANY, I. 2007. Random polytopes, convex bodies, and approximation. In *Stochastic Geometry*, W. Weil, Ed. Lecture Notes in Mathematics Series, vol. 1892. Springer Berlin Heidelberg, 77–118.

BENTLEY, J. L., PREPARATA, F. P., AND FAUST, M. G. 1982. Approximation algorithms for convex hulls. *Commun. ACM 25,* 1, 64–68.

BRONSTEIN, E. 2008. Approximation of convex sets by polytopes. *Journal of Mathematical Sciences 153,* 6, 727–762.

CATTHOOR, F. 1999. Energy-delay efficient data storage and transfer architectures and methodologies: Current solutions and remaining problems. *J. VLSI Signal Processing 21*, 219–231.

CATTHOOR, F. ET AL. 1998. System-level transformations for low power data transfer & storage. In *Low-Power CMOS Design*. IEEE, USA, 609–618.

CHO, D. ET AL. 2007. Software controlled memory layout reorganization for irregular array access patterns. In *CASES*. ACM, USA, 179–188.

CLAUSS, P. ET AL. 2000. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News 28*, 11–19.

CLAUSS, P. ET AL. 2009. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *Trans. VLSI 17,* 8, 983–996.

CLAUSS, P., GARBERVETSKY, D., LOECHNER, V., AND VERDOOLAEGE, S. 2011. Polyhedral Techniques for Parametric Memory Requirement Estimation. In *Energy-Aware Memory Management for Embedded Multimedia Systems: A Computer-Aided Design Approach*, F. Balasa and D. Pradhan, Eds. Chapman & Hall/Crc Computer and Information Science. Taylor and Francis.

CLAUSS, P., GARBERVETSKY, D., LOECHNER, V., AND VERDOOLAEGE, S. 2012. *Polyhedral Techniques for Parametric Memory Requirement Estimation in Energy-Aware Memory Management for Embedded Multimedia Systems: A Computer-Aided Design Approach*. CRC Press, Chapter 4, 117–149. Energy-Aware Memory Management for Embedded Multimedia Systems: A Computer-Aided Design Approach.

COHEN, A. ET AL. 1999. Storage mapping optimization for parallel programs. In *Proc. Int'l Euro-Par*. Springer, London, UK, 375–382.

CONG, J. ET AL. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Trans. Des. Autom. Electron. Syst. 16,* 2, 15:1–15:25.

CREUSILLET, B. AND IRIGOIN, F. 1996. Exact vs. approximate array region analyses.

DARTE, A., SCHREIBER, R., AND VILLARD, G. 2005. Lattice-based memory allocation. *Computers, IEEE Transactions on 54,* 10, 1242–1257.

DE GREEF, E. ET AL. 1997. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Parallel Computing*. 84–98.

DONGARRA, J. ET AL. 1990. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *J. Parallel & Distributed Computing 9,* 2, 185–202.

DUDA, R. O. AND HART, P. E. 1972. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM 15,* 1, 11–15.

EDDY DE GREEF, F. C. AND DE MAN, H. 1996. Reducing storage size for static control programs mapped onto parallel architectures. In *Dagstuhl Seminar on Loop Parallelisation*. IEEE, USA, 728–735.

FEAUTRIER, P. 1988. Array expansion. In *Proceedings of the 2Nd International Conference on Supercomputing*. ICS '88. ACM, New York, NY, USA, 429–441.

FILIPPOPOULOS, I., CATTHOOR, F., AND KJELDSBERG, P. 2014. Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios. *Design Automation for Embedded Systems*, 1–24.

FILIPPOPOULOS, I., CATTHOOR, F., KJELDSBERG, P., HAMMARI, E., AND HUISKEN, J. 2012. Memory-aware system scenario approach energy impact. In *NORCHIP, 2012*. 1–6.

FRANSSEN, F. ET AL. 1993. Modeling multidimensional data & control flow. *VLSI 1,* 3, 319–327.

GRÖSSLINGER, A. 2009. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Proc. Int'l Conf. Compiler Construction*. Springer, Berlin, 236–250.

GUTHAUS, M. R. ET AL. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IISWC*. IEEE, USA, 3–14.

JANG, B. ET AL. 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *Trans. Parallel & Distributed Systems 22*, 105–118.

JANJUSIC, T. ET AL. 2011. Gleipnir: A memory analysis tool. In *Proc. ICCS*. 2058–2067.

JHA, P. ET AL. 1997. Library mapping for memories. In *Proc. EDAC*. IEEE, USA, 288–.

JUURLINK, B., ALVAREZ-MESA, M., CHI, C., AZEVEDO, A., MEENDERINCK, C., AND RAMIREZ, A. 2012. Understanding the application: An overview of the h.264 standard. In *Scalable Parallel Programming Applied to H.264/AVC Decoding*. SpringerBriefs in Computer Science. Springer New York, 5–15.

KANDEMIR, M. T. 2001. A compiler technique for improving whole-program locality. *SIGPLAN Not. 36,* 3, 179–192.

KJELDSBERG, P. ET AL. 2003. Data dependency size estimation for use in memory optimization. *IEEE TCAD 22*, 908–921.

KJELDSBERG, P. ET AL. 2004. Storage requirement estimation for optimized design of data intensive applications. *ACM TODAES 9,* 2, 133–158.

KRITIKAKOU, A. ET AL. 2013. Near-optimal & scalable intra-signal in-place for non-overlapping & irregular access schemes. *ACM TODAES 19,* 1.

KRITIKAKOU, A., CATTHOOR, F., KELEFOURAS, V., AND GOUTIS, C. 2013. A systematic approach to classify design-time global scheduling techniques. *ACM Comput. Surv. 45,* 2, 14:1–14:30.

KRITIKAKOU, A., CATTHOOR, F., KELEFOURAS, V., AND GOUTIS, C. 2014. A scalable and near-optimal representation of access schemes for memory management. *ACM Trans. Archit. Code Optim. 11,* 1, 13:1–13:25.

LEE, C. ET AL. 1997. Mediabench: a tool for evaluating & synthesizing multimedia & communicatons systems. In *Proc. Int'l Symp. Microarchitecture*. IEEE, Washington, USA, 330–335.

LEE, L. ET AL. 2007. An optimization model for storage yard management in transshipment hubs. In *Container Terminals and Cargo Systems*. Springer, Berlin, Heidelberg, 107–129.

LIPPENS, P. ET AL. 1993. Allocation of multiport memories for hierarchical data stream. In *Proc. CAD*. IEEE, USA, 728–735.

LUK, C.-K. ET AL. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not. 40,* 6, 190–200.

MAYDAN, D. E. ET AL. 1993. Array-data flow analysis and its use in array privatization. In *Proc. Symp. POPL*. ACM, NY, USA, 2–15.

MEISTER, B. AND VERDOOLAEGE, S. 2008. Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In *In ODES-6: 6th Workshop on Optimizations for DSP and Embedded Systems*.

NACHTERGAELE, L. ET AL. 1992. Specification and simulation front-end for hardware synthesis of dsp applications. *Int. J. Comp. Simulation 2*, 213–2291.

NETHERCOTE, N. ET AL. 2006. Building workload characterization tools with valgrind.

PAEK, Y. ET AL. 2002. Efficient & precise array access analysis. *TOPLAS 24,* 1, 65–109.

PALEM, K. ET AL. 2002. Design space optimization of embedded memory systems via data remapping. In *LCTES*. ACM, USA, 28–37.

PANDA, P. R. ET AL. 2001. Data and memory optimization techniques for embedded systems. *ACM TODAES 6,* 2, 149–206.

POUCHET, L.-N. ET AL. 2012. Polybenchmarks benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

RAMANUJAM, J. ET AL. 2001. Reducing memory requirements of nested loops for embedded systems. In *Proc. DAC*. ACM, USA, 359–364.

RUBIN, S. ET AL. 2002. An efficient profile-analysis framework for data-layout optimizations. *SIGPLAN Not. 37*, 140–153.

SEGHIR, R. ET AL. 2012. Integer affine transformations of parametric polytopes and applications to loop nest optimization. *ACM Trans. Archit. Code Optim. 9,* 2, 8:1–8:27.

SHEN, Z., LI, Z., AND YEW, P. 1990. An empirical study of fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems 1*, 356–364.

SO, B. ET AL. 2004. Custom data layout for memory parallelism. In *CGO*. IEEE, USA, 291–.

TRONON, R., BRUYNOOGHE, M., JANSSENS, G., AND CATTHOOR, F. 2002. Storage size reduction by in-place mapping of arrays. In *Verification, Model Checking, and Abstract Interpretation*, A. Cortesi, Ed. Lecture Notes in Computer Science Series, vol. 2294. Springer Berlin Heidelberg, 167–181.

VAN SWAAIJ, M. ET AL. 1992. Automating high level control flow transformations for dsp memory management. In *Proc. Work. VLSI Signal Processing*. IEEE, USA, 397–406.

VANBROEKHOVEN, P., JANSSENS, G., BRUYNOOGHE, M., AND CATTHOOR, F. 2005. Transformation to dynamic single assignment using a simple data flow analysis. In *Programming Languages and Systems*, K. Yi, Ed. Lecture Notes in Computer Science Series, vol. 3780. Springer Berlin Heidelberg, 330–346.

VANBROEKHOVEN, P., JANSSENS, G., BRUYNOOGHE, M., AND CATTHOOR, F. 2007. A practical dynamic single assignment transformation. *ACM Trans. Des. Autom. Electron. Syst. 12,* 4.

VANBROEKHOVEN, P., JANSSENS, G., BRUYNOOGHE, M., CORPORAAL, H., AND CATTHOOR, F. 2003. A step towards a scalable dynamic single assignment conversion. Tech. rep.

VERDOOLAEGE, S. ET AL. 2013. Barvinok. http://barvinok.gforge.inria.fr/.

VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNOOGHE, M. 2007. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica 48,* 1, 37–66.

V.GHEORGHITA ET AL. 2009. System scenario based design of dynamic embedded systems. *ACM TODAES 14,* 3, 1–45.

WEIDENDORFER, J. ET AL. 2004. A tool suite for simulation based analysis of memory access behavior. In *Proc. ICCS*. Springer, 440–447.

WUYTACK, S. ET AL. 1998. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. *TVLSI 6*, 529–537.