A Scalable and Near-Optimal Representation of Access Schemes for Memory Management

ANGELIKI KRITIKAKOU, ONERA and University of Patras
FRANCKY CATTHOOR, IMEC and KU Leuven
VASILIOS KELEFOURAS and COSTAS GOUTIS, University of Patras

13

Memory management searches for the resources required to store the concurrently alive elements. The solution quality is affected by the representation of the element accesses: a sub-optimal representation leads to overestimation and a non-scalable representation increases the exploration time. We propose a methodology to near-optimal and scalable represent regular and irregular accesses. The representation consists of a set of pattern entries to compactly describe the behavior of the memory accesses and of pattern operations to consistently combine the pattern entries. The result is a final sequence of pattern entries which represents the global access scheme without unnecessary overestimation.

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: Storage size, near-optimality, scalability, iteration space, memory management, memory optimization, resources

ACM Reference Format:
Angeliki Kritikakou, Francky Catthoor, Vasilios Kelefouras, and Costas Goutis. 2014. A scalable and near-optimal representation of access schemes for memory management. ACM Trans. Architec. Code Optim. 11, 1, Article 13 (February 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2579677

1. INTRODUCTION

Memory management techniques search for the minimum required resources to store the data. They are applied in domains where the size is crucial, such as in the lower layers of on-chip memory hierarchy of systems (e.g., scratchpad memories of the embedded systems [Grösslinger 2009] and hardware controlled caches [Catthoor 1999]). The near-optimal computation of the resources is essential, as it directly affects the cost, area, and power consumption [Catthoor 1999]. For instance, many embedded systems have tight memory space constraints [Ozturk et al. 2008], whereas the memory units contribute to the cost [Panda et al. 1999], as the power is heavily dominated by array storage [Catthoor 1999].

Authors' addresses: A. Kritikakou, ONERA, 2 Av. douard Belin, 31000 Toulouse; F. Catthoor, IMEC, Kapeldreef 75, 3001 Leuven, Belgium; V. Kelefouras and C. Goutis, University of Patras, Department of Electrical and Computer Engineering, Patras, Greece, 26500; email: akritikakou@ece.upatras.gr. The results were co-financed by Public Welfare Foundation "Propondis" research funds and Greek National funds (Heracleitus-II).

DOI: http://dx.doi.org/10.1145/2579677

The computation of the required number of storage resources is based on the access scheme derived from the application under study (e.g., the loops, conditions statements, and memory access statements). The consistent combination of this information describes the global memory access scheme that defines the storage resources. In commonly used embedded system applications, such as image, video, and signal processing, arrays with very regular memory accesses are the dominated data inside loops with conditions. The conditions disturb the regularity of the memory accesses, creating "holes" in the iteration space (i.e., making parts of the iteration space invalid). When a high number of holes and several array access statements exist, the iteration space becomes highly complicated. Existing approaches describe the iteration space in an enumerative way, in a symbolic way, or by worst-case approximation, as explained in Section 3. The enumerative representations are optimal but not scalable, as the exploration time is increased in the increase of the number of memory accesses. The symbolic representations are scalable and near-optimal up to quite regular iteration spaces. When applied in irregular spaces, they either have to approximate the space by, for example, applying a convex hull, or they split the space into potentially too many regular parts, increasing the exploration time, as no control over the splitting process exists. The approximations consider the invalid parts as valid leading to resources overestimation. Hence, a near-optimal and scalable representation is highly desired.

The proposed representation is scalable and near-optimal for complex iteration spaces with both regular and irregular holes created by memory access statements in loop structures with manifest conditions. Our first contribution is the concept and the formulation of patterns. A pattern represents the iteration space of a statement through a compact and repetitive description avoiding enumeration. A pattern is defined by a sequence of segments. In each segment, the first value is the number of consecutive iterators, where the statement has the same behavior to the memory. The second values is the actual behavior—that is, Access (A) or Hole (H). By including the holes, the invalid iteration space parts are described avoiding suboptimal approximations and the bear-optimality is controlled. For example, {1H 1A} repeated five times represents a statement, which accesses only in the odd iterators from 0 to 10. The second contribution is the proposed pattern operations required to consistently combine the patterns in the iteration space under all possible cases of our target domain. The third contribution is the application of the proposed representation in the intrasignal in-place optimization step to compute the maximum number of concurrent alive elements, which is the minimum resources required to store an array. We also demonstrate and evaluate the proposed representation for several benchmarks from the Polybench [Pouchet et al. 2012], the Mibench [Guthaus et al. 2001], and the Mediabench [Lee et al. 1997] suites.

The article is organized as follows. Section 2 motivates the proposed representation. Section 3 presents existing representations used in storage size management approaches. Section 4 describes the problem and the target domain. Section 5 summarizes the proposed representation. Section 6 describes the pattern formulation, Section 7 presents the pattern operations, and Section 8 reviews the intrasignal inplace computation. Section 9 evaluates our study, and Section 10 concludes this study.

2. MOTIVATION

When the memory access statements are regularly executed, the iteration space is solid. A condition disturbs this regularity by potentially introducing a high number of holes. When several conditions coexist, the iteration space becomes significantly complex and quite irregular. To illustrate the aforementioned problem, we use the simple example of Figure 1. The initial part of the access scheme up to k=6 and i=24 is depicted in Figure 1(a). The black dots describe the iterations where the array

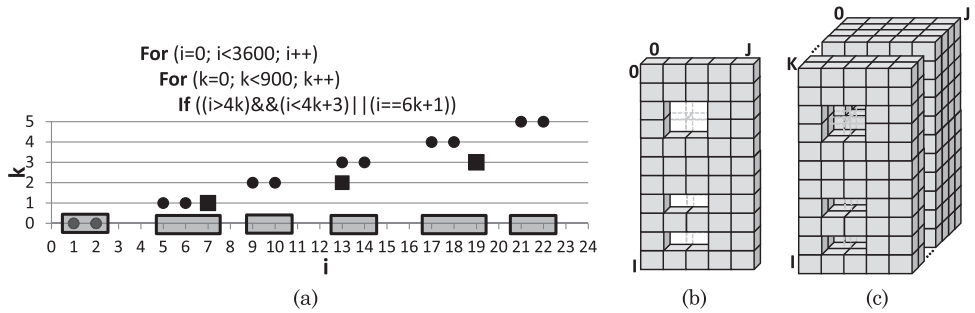


Fig. 1. (a) 1D case: code and iteration space. The black dots and squares describe the iterations that access the array for C1 and C2. The gray cells are the elements that are accessed each time. The gray cells describe the accessed elements for 2D (b) and 3D (c) case.

elements are accessed due to condition C1: $(i > 4k)\&\&(i < 4k + 3)$ and the black squares due to condition C2: $i == 6k + 1$. The gray boxes show the corresponding accessed elements. The enumeration approaches are not scalable, when the number of memory accesses is increased (e.g., high loop bound). If only C1 condition exists, the access shape is repetitive (i.e., the projection of the dots in i axis). By adding C2 condition through the OR operation, it disturbs the repetition of initial access shape. The symbolic approaches have to increase the number of constraint linear equations to describe the space. In contrast, the proposed methodology represents the first and second condition by pattern {2A 2H} repeated 900 times and the third condition by pattern {1A 5H} repeated 600 times. After applying the pattern operations, the final pattern is {2A 2H 3A 1H 2A 2H} repeated 300 times. The maximum number of concurrent alive elements is given by the summation of the accessed parts multiplied by the repetition factor (i.e., $(300) * (2 + 3 + 2) = 2100$). By increasing the array dimensions, the symbolic approaches become less efficient in highly irregular spaces. In Figure 1(b), the symbolic approaches approximate the accessed region by applying a convex hull approximation—that is, the holes of size 2×2 and 1×2 are considered as accesses and the computed size is 55. The alternative for a near-optimal computation is to split the iteration space to at least six polytopes, which increases the number of linear equations required to describe the iteration space. It can be readily extrapolated that this will further increase for larger irregular spaces. In contrast, the proposed representation in the I dimension uses the {2A 2H 3A 1H 1A 1H 1A} pattern and in J the {1A 2H 2A} pattern. The size is computed by propagating the size of the outer dimension to the inner dimension. The propagation multiplies the size of the I dimension with the size of loop J (i.e., $7 * 5$) and explores potential new elements due to J dimension. The latter is given by multiplying the holes in the I dimension by the size derived from the J pattern (i.e., $4 * 3$). The result is 47. When another dimension is added in Figure 1(c), the size loss due to the approximation is multiplied by the size of the third loop (e.g., in our example, the size loss of 8 is multiplied by the size of k dimension).

3. RELATED WORK

Existing storage management techniques, such as those for evaluation of the working set size or for memory mapping, represent the iteration space by enumeration or symbolic approaches. The enumeration representation is optimal but not scalable in the number of memory accesses. The symbolic representation is near-optimal and scalable up to quite regular iteration spaces. However, when several irregular holes exist, the symbolic approaches either approximate through convex hull or have to potentially

highly increase the number of linear equations to describe the space, as they miss a control mechanism over the polytopes splitting process.

Enumerative representations are reference list–based schemes used to describe each array reference without any summarization [Paek et al. 2002]. For instance, background memory size estimation is performed by enumerating the indexed signals of all index expressions [Nachtergaele et al. 1992]. In So et al. [2004], a custom memory data layout is proposed to parallelize memory accesses based on the overall access pattern. Several memory management techniques use profiling and instrumentation tools to enumerate the memory accesses. Palem et al. [2002] select candidates for data remapping through memory access patterns along program hot-spots. Rubin et al. [2002] search the space with all possible memory data layouts by iteratively prototyping and evaluating candidate data layouts. Panda et al. [2001] present techniques for memory data layout. A profiling-based strategy generates an access trace and exploits through a heuristic the scratchpad memory hierarchy in Cho et al. [2007]. Several instrumentation and profiling tools provide the memory accesses enumeration. For example, SHMAP [Dongarra et al. 1990] and Gleipnir [Janjusic et al. 2011] collect memory access traces. Pin [Luk et al. 2005] provides an instrumentation platform to trace memory instructions size. MemSPy [Martonosi et al. 1992] analyzes by simulation the memory accesses. Although the enumerative approaches are optimal, when accesses are increased, the exploration time is prohibited.

An alternative representation describes the memory accesses using symbolic forms—for example, linear constraint-based schemes where accesses are expressed as convex regions in a geometrical space [Paek et al. 2002]. Quality is reduced when nontrivial array reshaping is performed at boundaries, as the whole array is assumed to be accessed, when regions must be widened to form a convex hull and when nonaffine expressions are used, which removes the corresponding constraint equations [Paek et al. 2002]. Simplified constraint-based forms (e.g., Balasundaram and Kennedy [1989]) describe Solid Iteration Spaces (SIS) (e.g., diagonal or triangular shapes) but are less efficient for shapes with holes. Triple notation [Shen et al. 1990] (i.e., lower bound, upper bound, and stride per dimension) has been used to describe regular spaces. Polytope theory is very commonly used for regular spaces—for example, the space is represented by placing polytopes of signals in a common place with ILP techniques [Kjeldsberg et al. 2003]. IMEC Atomium [Catthoor et al. 1998b] supports memory-related steps based on the polyhedral dependency graph [Swaaij et al. 1992]. The Data Transfer and Storage Exploration (DTSE) methodology uses the polyhedral dependency graph to explore several substeps relevant to the memory data layout optimization step—for instance, a two-stage heuristic DTSE approach in Wuytack et al. [1997], the platform independent DTSE optimization step applied for a parallel cavity detection algorithm in Danckaert et al. [1999], and a polytope data access graph of the array memory operations used as input to the data reuse exploration in memory hierarchy design step in Wuytack et al. [1998]. Estimation of storage requirements with a partial fixed ordering through polytopes is proposed in Kjeldsberg et al. [2004]. Philips Phideo [Lippens et al. 1993] is oriented to stream-based video applications and represents the space as linear function of the iteration index. Jang et al. [2011] use memory access vectors, the loop nest depth, and the array dimension. In Kandemir [2001], an access matrix describes the array accesses to explore data locality. Clauss and Meister [2000] focus on spatial locality optimization using utilization vectors for references. Cong et al. [2011] use polytopes for memory partition and scheduling. Distance vectors with data access matrices are used for uniform references [Ramanujam et al. 2001]. However, the symbolic approaches for solid spaces are not directly applicable in spaces with holes.

Some symbolic approaches have been extended to piecewise regular iteration spaces—that is, spaces with few holes among polytopes. However, they are less

efficient and less scalable for iteration spaces with increased number of irregular holes, as they cannot efficiently represent them. For instance, SUIF [Maydan et al. 1993] and PIPS [Creusillet and Irigoin 1996] add additional constraint to the representation in order to deal with a hole. As the holes are increased, the number of linear expressions is also increased [Paek et al. 2002]. In Van Achteren and Lauwereins [2000], the data-reuse exploration methodology is extended for holes in the signal access patterns represented by the data access polytope graph. In Franssen et al. [1993], piecewise linear and constant modulo indexing are handled by extending the node space and transforming the spaces to a representation suitable for polyhedral graph. In Balasa et al. [1994], a transformation is presented for modulo expressions to affine expressions. In Darte et al. [2005], lattices represent the iteration space for the memory allocation problem. Seghir et al. [2012] propose a lattice intersection to count the integers in Z-polytopes. When the iteration space cannot be efficiently represented by lattice, the aforementioned approaches either approximate the space by solidifying holes or split it, leading to an increased number of linear equations and search time.

In this work, we propose a scalable and near-optimal representation for complex iteration spaces with both regular and irregular holes, as shown in the next sections.

4. PROBLEM FORMULATION AND TARGET DOMAIN

4.1. Problem Formulation

Several design steps apply techniques to compute the size required to store the elements of the arrays. For instance, the DTSE methodology of Catthoor et al. [1998a] exploits the memory organization of embedded systems for the lower memory layers through a series of steps—for example, data reuse and related data copy decisions, memory hierarchy level assignment, memory allocation and assignment, intrasignal in-place optimization, and intersignal in-place optimization. They are especially applied in the lower layers of the on-chip background memory hierarchy, where size is crucial. The lower memory layers are quite close to the processing part of the system and should be relatively small in size to provide fast access to the stored elements. The proposed representation and the optimization technique considered in this study are less relevant for off-chip bulk memories and long-term storage, such as hard disk drives, nonvolatile memories, flash-based or future MRAM-based solid-state storage. In this context, near-optimality of the techniques is crucial, as a suboptimal result leads to overestimation of the required resources. Then, either the on-chip memory size is increased, which increases area, cost, and power consumption, or the array elements are spilled to higher memory layers, which degrades performance due to the memory misses. Hence, a tight (near-optimal) computation of the required resources is crucial for the system design. Scalability of the techniques is of great importance, since they are applied in industry applications, which involve substantial amount of code with highly increased number of memory accesses. As time-to-market is crucial, the increase in the number of accesses should not lead to prohibited exploration time for the relevant design steps (beyond a few hours of CPU time). Otherwise, the design process and the final design are delayed.

After the memory optimization steps are applied, the result is used by the address generation design step. An efficient addressing is achieved, when the addresses are mainly regular. A high irregularity increases the time of address generation and element accessing. For instance, the address generation and accessing of sequential elements is very efficient (e.g., when the memory burst mode is used). Hence, the memory optimization result should not introduce too much irregularity and still maintain its near-optimality. To achieve a near-optimal and scalable memory optimization technique that supports efficient addressing for the different memory organization

design steps, a representation is required for the memory access scheme, which meets near-optimality, scalability, and addressing constraints, which is the goal of this article.

The proposed representation is applied in the context of intrasignal in-place optimization step for nonoverlapping stores and loads. The intrasignal in-place result is the size required to store the elements of each individual array [Catthoor et al. 1998a]. The intrasignal in-place computes the array window—that is, the maximum number of concurrent alive elements during the application execution. The stores and the loads are nonoverlapping when the loads are executed in later loop iterations after all of the stores have been executed. As the stored array elements are valid until the first load is executed, the maximum number of concurrently alive elements exists after the execution of all stores and before the execution of any load. As the array window computation is used by next memory optimization steps, the near-optimality of the result is important, since unnecessary overestimations should be avoided. For instance, the array window is used by the intersignal in-place optimization step to compute the near-optimal location of the arrays in the memories so that the reuse of memory locations between arrays is maximized, and then by the mapping to the physical memories [Catthoor et al. 1998a]. Code transformations to improve the global data and control flow or the data reuse (e.g., array restructuring in Leung and Zahorjan [1995]) have been already applied upfront, as they belong to the early steps in the overall memory management methodology (e.g., as explained in motivation in Catthoor et al. [1998a]).

4.2. Target Domain

The target domain consists of applications with deterministic behavior. If several threads are accessing the same array, all memory accesses and condition statements are considered by the representation. The applications are highly loop dominated and data dominated (i.e., a high percentage of the executed code handles indexed array signals in the context of loops)—for instance, high speed data intensive applications in the fields of speech, image, and video processing, which require significant amount of storage [Jha and Dutt 1997]. Due to the way of writing the code, that is, the enumerative conditions that describe a part of the iteration space are few and the parametric conditions are used to describe several parts, a finite and, usually, small set of manifest condition statements exists in the application. The arrays are statically allocated. When data-dependent applications exist with nonmanifest conditions and dynamically allocated data, preprocessing is required. The system scenario approach [Gheorghita et al. 2009] can be applied to convert the application into a set of individual system scenarios that have manifest code internally. Then, the proposed representation is applied per scenario. For dynamically allocated data, Dynamic Data Type Refinement (DDTR) [Bartzas et al. 2006] and Dynamic Memory Management Refinement (DMMR) [Atienza et al. 2006] can be applied to convert dynamically allocated data into bounded virtual memory segments, which can be treated as an array segment by our approach. The applications of the target domain are described by a unified template, which is used to describe the input to the proposed representation in a unified way. The unified template has one to many loop nests and uses single assignment code—that is, an array element is written once, although read several times. The index expressions of the access statements are of "iterator+constant" (e.g., $i + b$) type, since it is a highly occurring case, especially in multimedia applications such as those in Pouchet et al. [2012], Lee et al. [1997], and Guthaus et al. [2001]. The selected index expression allows a better explanation of the proposed principles. The "coefficient*iterator+constant" (e.g., $a * i + b$) index type is translated into "iterator+constant" index type and a set of parametric manifest conditions, which describe the holes in the iteration space. For example, the index expression $2i + 1$ is expressed as index expression i + 1 and a parametric condition $i == 2k$. Index expressions with modulo operation on the iterator are expressed as

"iterator+constant" index type and an extra loop with size equal to the constant (e.g., *i%4* inserts one k loop from 0 to 4), the index expression becomes k or translated to linear expressions by extending dimensions [Franssen et al. 1993].

The unified template consists of primitive conditions and primitive operations to describe condition expressions. The conditions can be enumerative using constants and parametric conditions using linear expressions and can describe an SIS, where sequential array elements are accessed between iterations, and an Iteration Space with Holes (ISH). We define the primitive conditions as follows:

Definition 4.1. The primitive conditions are:

(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—that is, $i < a$, $i > a$, $(i < a)\&\&(i > b)$ and $i == a$.
(2) Enumerative Conditions for iteration space 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—that is, $i < c * k + d$ and $i > c * k + d$. For one-dimension arrays and multiple loop structures, the PCS can be mapped to the ECS of $i < ((c * UB_{\mathrm{k}} - 1) + d) + 1$.
(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—that is, $i == c * k + d$, $i \neq c * k + d$ and $(i > c * l + d_1)\&\&(i < c * k + d_2)$ (with $d_1 < c$ and $d_2 < c$).

When the ECS/PCS are combined with the || primitive operator, they create ECH/PCH. If the conditions under study use another comparison operator, they are mapped to the primitive conditions. For instance, the ECS condition $i \geq LB$ is mapped to $i > LB - 1$ (e.g., $i \geq 5$ is mapped to $i > 4$), the ECH condition $i \neq d$ is mapped to $(i < d)||(i > d)$ (e.g., $i \neq 3$ is mapped to $(i < 3)||(i > 3)$), the PCS condition $i \geq c * k + d$ is mapped to $i > c * k + d - 1$ (e.g., $i \geq 2 * k$ is mapped to $i > 2 * k - 1$), and so forth.

Definition 4.2. The primitive operations are OR and AND operations.

The remaining logical operations can be expressed by OR and AND operations. For instance, $(i == a)\,NAND\,(j == b)$ is transformed to $(i \neq a)||(j \neq b)$.

5. PATTERN-BASED REPRESENTATION: SYNOPSIS

The proposed pattern-based representation avoids the enumeration of the valid iteration parts, as it symbolically describes a high number of parts. The proposed representation consists of pattern entries and pattern operations over the pattern entries. A pattern entry consists of a pattern and a set of parameters, as described in detail in Section 6. The pattern operations are applied over the pattern entries to consistently combine them in the iteration space, as described in Section 7.

Definition 5.1. The pattern-based representation consists of a finite set of pattern entries and pattern operations used as combination methods over the pattern entries.

Initially, the application under study is mapped to the unified template, which is instantiated—that is, the parameters of the template take specific values. In the first step, the instantiated unified template is parsed to create a primitive pattern entry per condition/access statement per dimension. In order to describe the global memory access scheme, the pattern entries have to be consistently combined—that is, pattern entries that are referring to the same iteration space have to be considered together.

To achieve the consistent pattern combination, we define a set with the different cases that may occur in the unified template, as explained in Section 7.1. Then, we provide a scalable and near-optimal operation per combination case to systematically combine the pattern entries, described in Section 7.2. Using the primitive operations of the instantiated unified template and by applying the pattern operations, the partially/fully overlapping pattern entries are combined and the nonoverlapping pattern entries are virtually concatenated. The final result is a sequence of nonoverlapping pattern entries that describes the global memory access scheme.

Compared with existing approaches, the proposed representation has a mechanism to control the execution time and the near-optimality. When the holes of the new pattern entries are too small to provide a gain in the quality of the intrasignal in-place array window, although they introduce a high address generation cost due to increased irregularity, they are allowed to be considered as "virtual" accesses. In contrast to existing approaches, the position and the number of virtual accesses in the final sequence of patterns is controlled by the decision over the length of the holes that can be considered as virtual accesses. The smaller this value, the more close to optimal is the representation.

In realistic contexts, the combination of patterns is applied in a finite small set of pattern entries, since they are describing the iteration space in a repetitive way. The initial number of pattern entries is defined by the number of conditions and access statements. During operations, pattern entries are split and combined. If the number of pattern segments is larger than an acceptable value, holes can be considered as virtual accesses, or if the pattern has low repetition factor, it can be considered as virtual accesses. Then, pattern segments are merged and the number of the segments is decreased. In this way, a controllable approximation is applied whenever it is required to decrease the exploration time.

As we present the proposed representation in the context of nonoverlapping intrasignal in-place optimization, the maximum number of concurrent alive elements derives from the regions that are accessed, as described in Section 8. The accessed regions are described by the final sequence of the patterns per dimension and on the valid intrasignal in-place case. In the one-dimension case, this result derives by the adding the size of the pattern segments with A behavior multiplied by the repetition factor in the final pattern sequence. In multiple dimensions, the representation is applied per dimension, but the maximum number of concurrent alive elements derives by propagation of the sizes from the outer dimension to the inner dimensions. The propagation between dimensions depends on the intrasignal in-place case that is valid, which is decided based on the type of the final pattern entries in the representation sequence [Kritikakou et al. 2013a, 2013b]. As this work focuses on the pattern-based representation, we describe the pattern entries and the operations and illustrate them in the intrasignal in-place step for the one-dimension case, as the representation is applied per dimension.

6. PATTERN FORMULATION

We provide a list with the abbreviations used within this document in Table I.

The valid and the invalid part of iteration space defined by a primitive condition or an access statement is described by a pattern entry—that is, a pattern with a set of parameters. By including the invalid part (i.e. the holes), the iteration space can be described in a regular and repetitive way due to the loop structure to which the statement belongs.

Definition 6.1. A pattern is a sequence of segments, where a segment is described by (1) the number of consecutive iterator values (Segment Iterator Range [SIR]), where the statement describes the same type of behavior to the array, and (2) the type of the

Table I. Table Summarizing the Abbreviations

Term	Definition	Term	Definition
SIR	Segment Iterator Range	IR	Iterator Range
ST	Segment Type	PS	Pattern Size
LB	Lower loop Bound	UB	Upper loop Bound
LCM	Least Common Multiple	GCD	Greatest Common Divisor
SIS	Solid Iteration Space	ISH	Iteration Space with Holes
ECS	Enumerative Conditions for solid iteration Space	UF	Unrolling Factor
ECH	Enumerative Conditions for iteration space with Holes		
PCS	Parametric Conditions for solid iteration Space		
PCH	Parametric Conditions for iteration space with Holes		

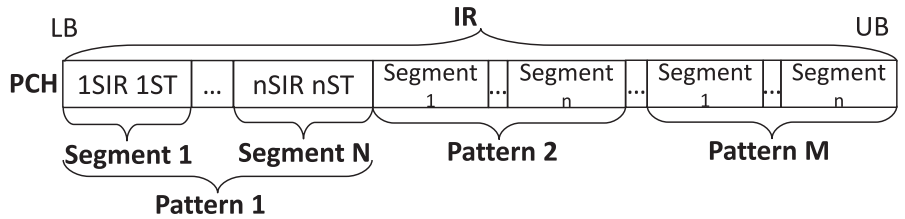


Fig. 2. The pattern consists of N segments, each segment has an SIR and an ST, the pattern is valid from *LB* up to *UB,* and it is repeated M times.

behavior (Segment Type[ST]), which is Access (A) or Hole (H)—that is, 1SIR 1PT ... nSIR nPT.

Definition 6.2. The pattern parameters are:

(1) *Lower loop Bound* (*LB*) is the iterator value before the pattern begins. For the ECS/PCS and ECH conditions ($I > a$), the LB is $max(LB_{\mathrm{I}}, a)$, and for PCH conditions ($I > c * K + d$), it is $max(LB_{\mathrm{I}}, c * (LB_{\mathrm{K}} + 1) + d - 1)$.
(2) *Upper loop Bound (UB)* is the iterator after the pattern ends. For the ECS/PCS and ECH conditions ($I < b$), the *UB* is $min(UB_{\mathrm{I}}, b)$, and for the PCH conditions ($I < c * K + d$), it is $min(UB_{\mathrm{I}}, c * (UB_{\mathrm{K}} - 1) + c + 1)$.
(3) *Iterator Range (IR)* is the range of the iterator values where the pattern is valid (i.e., $UB - LB - 1$).
(4) *Pattern Size (PS)* is the summation of the lengths of all segments (i.e., $\sum_{i=1}^{Segments} iSIR$).
(5) *Repetition factor (R)* gives the times the pattern is repeated in the IR (i.e., $\frac{IR}{PS}$).
(6) *Iterator difference (Diff)* is the array subscript index difference between the WR and the RD access statements.

Figure 2 shows a pattern that consists of N segments, and each segment has an SIR and an ST. The pattern is repeated M times starting from the *LB* and terminates at *UB*.

We present several representative examples to illustrate the primitive pattern formulation:

(1) *Access statement:* The primitive pattern consists of one part with SIR equal to the *IR* and ST of A type. For instance, in Figure 3(a), the A behavior is valid for all iterations of the loop structure. As the loop bounds are $LB = 2$ and $UB = 10$, the *IR* is 7 and the pattern is {7A} with repetition factor $R = 1$.
(2) *ECS condition:* The pattern is similar to the previous example, but it has reduced SIR due to the condition. For instance, in Figure 3(b), the *LB* is derived from the maximum value between the condition expression and the *LB* of the loop

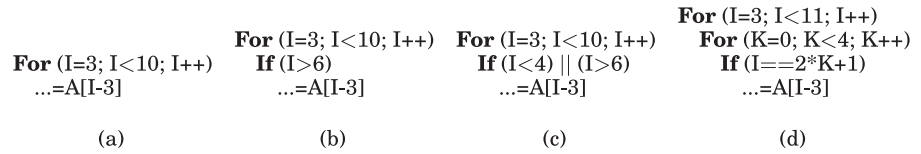


Fig. 3. Application code examples: access statement (a), ECS (b), ECH (c), and PCH (d).

structure—that is, $LB = max(LB_I, a) = max(2, 6) = 6$, the UB is 10, the IR is 3, the PS is 3, the $R = 1$, and the pattern is {3A}.

(3) *ECH condition:* The ECH pattern derives from the combined ECS conditions through the pattern operation. For instance, in Figure 3(c), the condition expression $(i < 4)||(i > 6)$ creates two pattern ECS entries—that is, {1A} with $LB = 2$, $UB = 4$, $IR = 1$, $R = 1$ and {3A} with $LB = 6$, $UB = 10$, $IR = 3$, and $R = 1$. By combining the pattern entries, the result is {1A 3H 3A} with $LB = 2$, $UB = 10$, $IR = 7$, and $R = 1$.

(4) *PCS condition:* The pattern is similar to the ECS condition, but with a repetition factor larger than one and length of the segment equal to the pattern size.

(5) *PCH condition:* The pattern depends on the conditions, as the PS is given by the coefficient of the linear expression and the comparison operator describes the type of the segment. For instance, in Figure 3(d), the PS is 2, LB of the pattern is $max(2, (2 * 0 + 1 - 1) = max(2, 0) = 2$, and the UB is $min(11, (2 * 3 + 2 + 1) = min(11, 9) = 9$. The $IR = 6$, $PS = 2$, $R = 3$, the first segment has length 1, and A as ST due to the == PCH type. The second segment is $PS - 1$ with ST equal to H—that is, {1A 1H}. If the UB of iterator K is increased to 5, the pattern is {1A 1H} with $R = 5$, since the $UB = 11$.

In realistic applications, several control statements with complicated conditions and access statements exist creating complex spaces. We discuss how our methodology handles this in the next sections.

7. DEFINITION OF PATTERN OPERATIONS

In order to find the global memory access scheme, the patterns entries of the primitive conditions and the access statements should be consistently combined. This section describes the different cases under which the patterns entries can be combined, described in detail in Section 7.1, and defines scalable and near-optimal pattern operations per combination case, described in Section 7.2.

7.1. Pattern Combination Cases

The possible combination cases of the patterns derive from analyzing the pattern formulation and the target domain. Figure 4 depicts the pattern combination cases (white boxes) and the corresponding pattern operations (gray boxes). In the next paragraphs, the combination cases (text in bold) are described and linked with the corresponding pattern operations (text in italic).

The first split is the overlapping or nonoverlapping patterns. The *overlapping patterns* have segments that are referring to the same iterators and, potentially, describe different access behavior. For instance, the patterns in Figure 3(a) ({7A} from $i > 2$ until $i < 10$) and in Figure 3(d) ({1A 1H} for $i > 3$ until $i < 9$) are overlapping. The *nonoverlapping patterns* describe the access behavior for different iterators. The nonoverlapping patterns may be *sequential*—that is, one pattern describes up to the iterator value x and the other pattern describes from $x + 1$ and so on. For the *nonsequential* patterns, iterator values exists between them, which are not described by either pattern—that is, one pattern describes up to the value x and the other pattern from $x + y$. The *sequential nonoverlapping* and the *nonsequential nonoverlapping* operations are described in

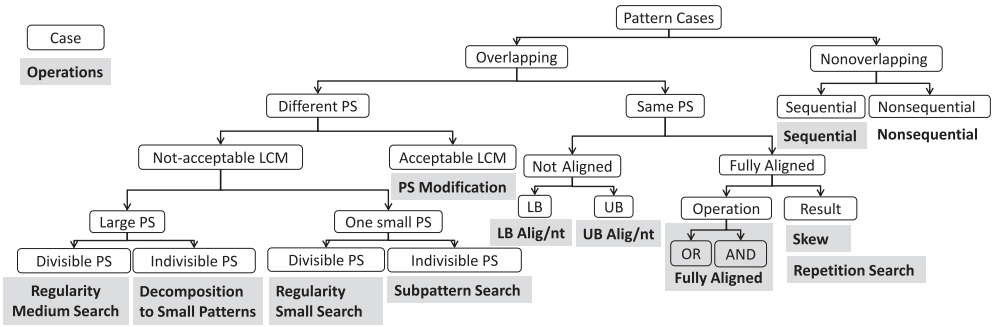
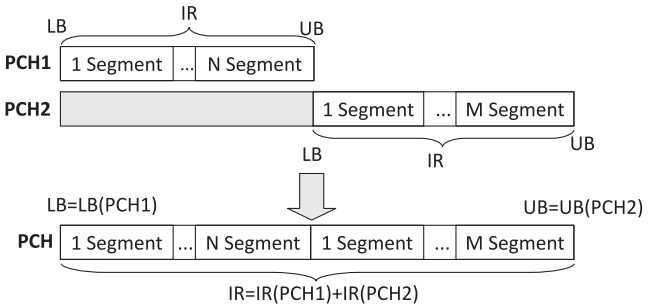


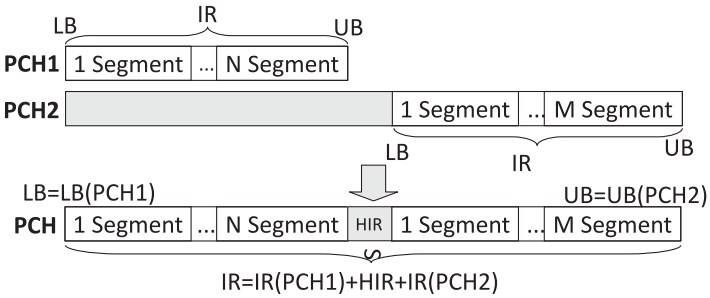
Fig. 4. Set of possible pattern combination cases with the corresponding operations.

Section 7.2.1. The overlapping patterns are further divided into the cases with same or different *PS*. In case of *same PS*, the patterns may be *fully aligned* or *not aligned*. When the patterns are fully aligned, the *LB* and *UB* of the patterns are equal and the primitive operations can be safely applied. The primitive operation can be an *OR operation*, which is explained in Section 7.2.2, or an *AND operation*, which is described in Section 7.2.2. The result may require postprocessing due to the existence of *sequential segments of same behavior* or *repetition of a smaller pattern* reducing the number of pattern segments. The sequential segments of same behavior in the resulting pattern are merged during OR/AND primitive operation execution. The merging of the first segment and the last segment with same behavior is explored by the *skewing operation*, which is described in Section 7.2.3. For instance, the PCH={3A 4H 3A} is modified to PCH′={4H 6A}. The repetition of a smaller pattern is explored by *repetition search* operation, described in Section 7.2.4. If PCH={3A 4H 3A 4H} with $R = 10$, the PCH is modified to PCH′={3A 4H} with R=20. When the patterns have the same PS and are *not aligned*, *LB* or/and *UB misalignment* may exist. The *LB alignment* operation, described in Section 7.2.5, is applied first. Then, the *UB alignment* operation is applied, described in Section 7.2.5.

When the patterns have *different PS*, operations for modifying the pattern size are applied. The Least Common Multiple (LCM) of the PS of the patterns is computed—for example, when $PS1 = 3$ and $PS2 = 4$, LCM is 12. The LCM is marked as acceptable when the modified *PS*, after unrolling based on the LCM, is quite low and, thus, not close to enumeration of the iteration space. If the *LCM is acceptable*, a *PS modification* operation, described in Section 7.2.6, is applied, which partially unrolls the patterns to LCM size. If the *LCM is not acceptable* due to the number of the LCM pattern segments and the LCM pattern size of the application under study, the OR and AND operations should be performed avoiding enumeration. We propose search operations based on the relative *PS* of the patterns. If one pattern has a *relative small PS* comparing with the other pattern, two cases exist: the small pattern is a factor of the *PS* of the other pattern (*Divisible PS*) or the *PS* of the patterns are *Indivisible*. In the first case, the *regularity small search* operations of Section 7.2.9 are applied to search for subpattern repetition. In the latter case, the *subpattern search* operations of Section 7.2.7 apply an iterative partial OR between the small pattern and a segment of the other pattern and search for repetition in the result. When both patterns have *large PS*, the *PS* of one pattern (medium pattern) may be a factor of the *PS* of the other pattern (*Divisible PS*) or the *PS* of the patterns are *Indivisible*. In the first case, *regularity medium search* operations of Section 7.2.10 are applied to search for pattern repetition. The *decomposition operation to small patterns* of Section 7.2.7 is introduced to split the medium pattern to smaller patterns.



(a) Sequential OR Operation



(b) Non-sequential OR Operation

Fig. 5. Result of nonoverlapping operations.

7.2. Pattern Operations

The next sections describe the pattern operations. The operations' execution time depends on the number of pattern segments n and not on the size of the application loops, as happens with the enumeration approach. When the application code is written, the designers do not fully unroll the large loops and use parametric conditions to express several holes in space. As the number of primitive pattern entries is defined by the number of condition and access statements in the loop kernel, it cannot be significantly increased. Since the pattern operations over the pattern entries allow only low unrolling factors to occur, the number of pattern segments will never increase to unacceptable values due to the operations. In addition, the control mechanism of the proposed methodology over the near-optimality ensures that the number of pattern segments cannot be uncontrollably increased.

7.2.1. Nonoverlapping Operations. Nonoverlapping operations "virtually" join two patterns (PCH1, PCH2) that refer to different iterators. It is virtual because the operation does not modify the pattern entries and orders them in a sequence of patterns. The *LB* and *UB* determine that the patterns are nonoverlapping. The OR primitive operation results to a pattern concatenation. The AND operation results to the null pattern due to the nonoverlapping case. As the patterns are not modified, near-optimality is maintained. The complexity is O(1). In the *Sequential Operation*, the *UB* of the first pattern is equal to the *LB* of the second pattern. The OR operation results to virtual pattern concatenation depicted in Figure 5(a). In case the patterns are the same, one pattern entry is maintained with modified parameters. The *Nonsequential Operation* is a virtual concatenation of the two patterns and a new segment with holes, which is expressed in the operation by the difference of the loop bounds of the first and the second pattern, as depicted in Figure 5(b).

```
UB(PCH)=UB(PCH1)
LB(PCH)=LB(PCH1)
PS(PCH)=PS(PCH1)
MPS=0
While (MPS≠PS(PCH))
  If (SIR(PCH1)==SIR(PCH2))
    SIR(PCH)=SIR(PCH1)
    If (PT(PCH1)==A||(PT(PCH2)==A)
      PT(PCH)=A
    Else
      ST(PCH)=H
  Else
    If (SIR(PCH1)>SIR(PCH2))
      max=PCH1
      min=PCH2
    Else
      max=PCH2
      min=PCH1
    If (PT(max)==A)
      PT(PCH)=A
      SIR(PCH)=SIR(max)
      size=SIR(max)-SIR(min)
      next SIR(min)=next SIR(min)-size
    Else
      SIR(PCH)=SIR(min)
      newSIR(max)=SIR(max)-SIR(min)
      If (PT(min)==A&&PT(max)==H)
        PT(PCH)=A
      Else
        PT(PCH)=H
  PCH=MPC|SIR ST
  MPS=MPS+SIR
  Next SIR(PCH1), SIR(PCH2)
```

Fig. 6. Pseudocode of fully aligned OR operation.

7.2.2. Fully Aligned Operations. Fully aligned operations are applied to patterns with the same *PS* and aligned bounds. The operations are applied segment by segment, thus the complexity is O(n). As the PS is acceptable, the type of the segments is not modified, maintaining near-optimality.

OR operation (||). The initial pattern entries (PCH1, PCH2) are removed from the representation, and the resulting pattern entry (PCH) is inserted. The bounds and the *PS* remain the same. The PCH depends on the type and the length of the initial segments. The operation is iteratively applied segment by segment until the PCH reaches the pattern size.

The pseudocode of the operation is described in Figure 6. In each iteration, an OR operation is applied between a segment of the first and of the second pattern. The segments length of the initial patterns is potentially modified, depending on the size of the dominant type segment. In case the lengths of the two segments are equal, the length of the segment in the resulting pattern is equal to SIR(PCH1). If the type of both initial segments is H, the type of the resulting segment is also H. Otherwise, it is A. When the lengths of the initial segments are different, the length of the result depends on the segment types. If both segments are of H type, the length is defined by the minimum segment length and the type is H. The segment with the large length is split into two segments: one with length equal to the small length of the initial segments and another segment with length equal to the difference of the lengths of the initial segments (i.e., large(SIR)-small(SIR)) used in the next iteration. This modification is depicted in Figure 7. The initial segments are depicted in Figure 7(a). Figure 7(b) shows the PCH2 with the larger length split into a segment equal to the segment length of PCH1 and a gray part left for the next iteration. If at least one segment has A type, the type of the result is A. The length of the result segment depends on which segment has PT=A. If the segment with the large length has A type, the length of the resulting segment is equal to the large length. Then, the pattern with the small length is modified: the SIR of the next segment is reduced by the difference of the SIRs, as depicted in Figure 7(c). The next segment of the PCH1 is reduced by the gray part. If only the segment with the small length is of A type, the segment length of the result is the small length. The segment with the large length is split into a segment with length equal to the small length for the current iteration and a part with length equal to the SIRs' difference. If the ST is equal to the ST of the previous PCH segment, the result

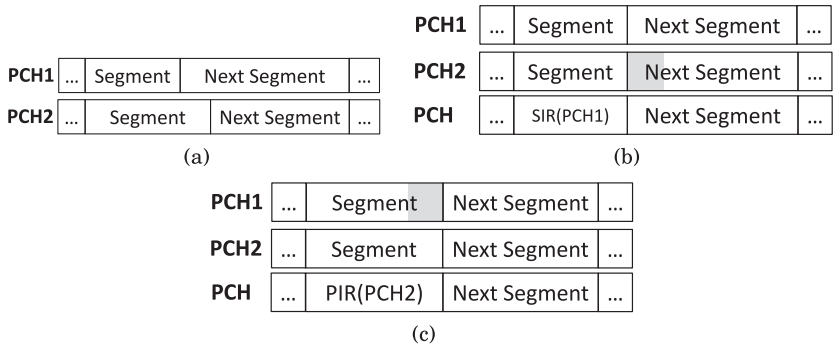


Fig. 7. Fully aligned OR operation. (a) Two segments of the PCH1 and PCH2. (b) When ST of both segments is H or the segment with the smaller SIR is A, the SIR of the result is equal to the small SIR. A new next segment in the pattern with the larger SIR is created with SIR equal to the SIRs' difference—that is, large(SIR)-small(SIR). (c) When ST of both segments is A or the segment with the larger SIR has ST=A, the SIR of the result is equal to the larger SIR. The SIR of the next segment of the pattern with the small SIR is reduced by the SIRs' difference.

is stored in one segment, reducing the total number of pattern segments. The process is repeated for the next PCH1 and PCH2 segments.

AND operation (&&). The AND operation is similar to the OR operation, but with adapted modification of the segments and the types of the patterns. When the length of the segments of the initial patterns is the same, the length of the result segment is equal to the initial length. If the type of the PCH1 and PCH2 segments is A, the type of the result is A. If at least one of the initial segments has H type, the type of the result is H. If the lengths of the initial segments are different, the length of the result depends on the type. If both segments have A type, the result is the minimum segment length and the ST is A. The segment with the large length is split into (1) one segment with length equal to the small length and one segment with length equal to the lengths difference. When at least one of the initial segments has ST of H type, the type of the result is H. The length depends on which of the two initial segments has ST equal to H. If the large length is H, the result is the large length. The length of the next segment of the pattern with the small length is reduced by the difference of the SIRs. If only the small length is of H type, the result is equal to the small length. The segment with the large length is split into a segment with length equal to the small length and a segment with length equal to the SIR difference. If the ST of the PCH segment is equal to the ST of the previous PCH segment, the segments are stored as one segment. The process is repeated for the next couple of initial segments.

7.2.3. Skew Operation. This operation merges the first and the last segment of a pattern (PCH) when they have the same type. This results to reducing the number of pattern segments by increasing their length. The ST of the segments is not modified, maintaining near-optimality. The operation splits one instance of the pattern repetition and extracts the first segment. The PCH is modified by combining the last segment of the first instance with the first segments of the second instance, and so forth. An additional pattern with the remaining segments of the final instance of the initial patterns is created. As the segments are not traversed, the complexity is O(1). An example is Figure 8, where the new ECS pattern entry (ECS1) describes the first segment of the first instance of the pattern, the modified PCH (PCH') with the first and the last segments merged, and a new ECH (ECH2) that describes the remaining segments of the last instance of the initial PCH. The new patterns are given by:

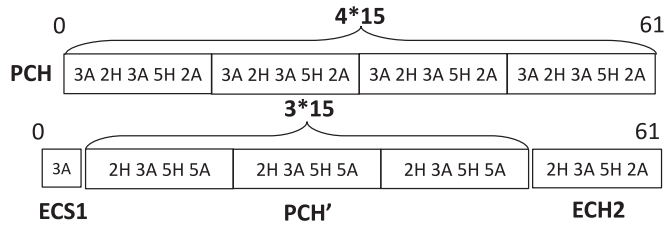


Fig. 8. Example of applying the skew operation. The initial PCH pattern is {3A 2H 3A 5H 2A}; after applying the skew operation, the ECS1, ECH2, and PCH' are created.

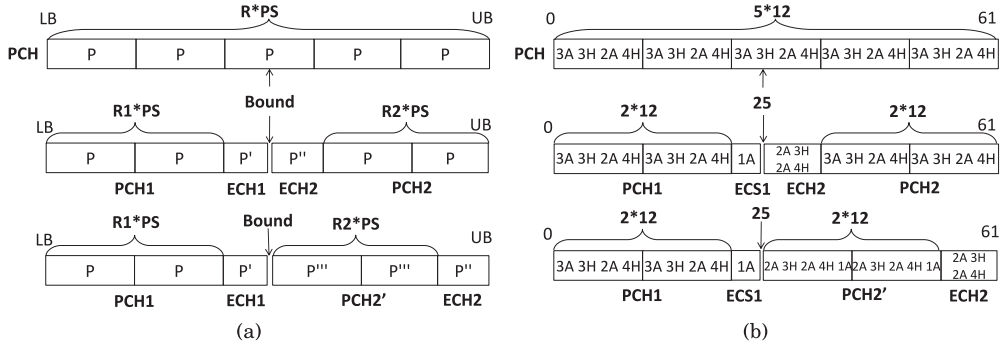
(1) *ECS1:* {1SIR 1PT}, $LB = LB(PCH)$, $UB = LB + 1SIR + 1$,
(2) *PCH':* {2SIR 2PT ... (NSIR+1SIR) NPT}, $LB = UB(ECS1) - 1$, $UB = UB(PCH) - \sum_{i=1}^{PCHParts} iSIR$, $R = R(PCH) - 1$, and
(3) *ECS2:* {2SIR 2PT ... NSIR NPT}, $LB = UB(PCH') - 1$, $UB = UB(PCH)$.

For instance, in Figure 8, the PCH is {3A 2H 3A 5H 2A} with $LB = 0$, $PS = 15$, $R = 4$, and $UB = 61$. After skew operation, the ECS1={3A} with $LB = 0$, $PS = 3$, $UB = 4$, the PCH' is {2H 3A 5H 5A} with $LB = 3$, $PS = 15$, $R = 3$, and $UB = 49$, and the ECS2= {2H 3A 5H 2A} with $LB = 48$ and $UB = 61$.

7.2.4. Repetition Search Operation. The repetition search operation searches the PCH to identify repetition of smaller patterns inside PCH to reduce the number of segments. The result is either a modified or not PCH pattern entry. The operation is applied in groups of segments and is repeated for a small number of times—that is, half the number of the pattern segments having O(nlog(n)) complexity. The potential new internal pattern is initialized with the first two segments. The remaining segments are compared in groups of two for potential repetition. The search continues by increasing the new internal pattern by one segment and comparing the remaining patterns in groups of three, and so forth. The operation terminates when the internal pattern has half the initial pattern segments. The segment type is not modified, maintaining near-optimality.

7.2.5. Alignment Operations. In alignment operations, the pattern is split into one pattern aligned with the bound and one nonoverlapping sequential pattern. The new patterns depend on where the bound cuts the pattern. As the operations are applied over the segments, they have O(n) complexity. The segments type is identical to the segments type of the initial pattern, maintaining near-optimality.

LB alignment. The operation may split the initial pattern (PCH) into four new pattern entries, depending on the new *LB* position. After the operation, the initial pattern entry is removed. When the bound cuts the PCH somewhere in the middle of the pattern and in half of the repetitions, two pattern entries with repetition (i.e., PCH1 and PCH2 in Figure 9(a)), and two pattern entries without repetition (i.e., ECH1 and ECH2 in Figure 9(a)) are created. The PCH is explored segment by segment to define the new patterns, thus the complexity is O(n). The PCH section on the left of the bound has size equal to $(bound + 1) - LB(PCH) - 1$. If this size is larger than two times the *PS*, a pattern with repetition (PCH1) occurs with lower bound equal to $LB(PCH)$, repetition factor equal to $(int)\frac{IR}{PS}$, and upper bound equal to $LB + R * PS + 1$. The cut of the PCH creates the ECH1 and ECH2 patterns. The PCH segments are traversed to find in which segment the bound cuts the pattern ($Part_{Bound}$). The ECH1 segments are the segments before the bound, and ECH2 has the segments after the bound. The ECH1 has *LB* equal to the maximum value of the $UB(PCH1) - 1$ and the $LB(PCH)$ and *UB* equal



(a) (b)

Fig. 9. LB alignment. (a) General case. The PCH is the initial pattern, and the bound is the new LB. The result creates the PCH1, ECS1, ECS2, and PCH2. The right section is skewed to align the PCH2 to the bound. (b) Example. The PCH is {3A 3H 2A 4H}, and the bound is 25.

to bound+1. To align the *LB* of the initial pattern, the ECH2 is moved after the PCH2 by skewing the PCH2. The skewed PCH2 pattern (i.e., PCH2' in Figure 9(a)) is {ECH2 ECS1}, the *LB* is equal to the bound, and the *UB* is given by $R2 * PS + bound + 1$. The ECH2 has *LB* equal to $R2 * PS + bound$ and *UB* equal to the *UB*(*PCH*). A degenerate case exists when the bound is a factor of the pattern size of the PCH. The PCH is split into PCH1 and PCH2 patterns with same segments. The PCH1 has *LB* equal to the *LB*(*PCH*), *UB* equal to bound+1, and repetition factor equal to $\frac{IR(PCH1)}{PS}$. The PCH2 has *LB* equal to bound, *UB* equal to the *UB*(*PCH*), and $R = \frac{IR(PCH2)}{PS}$. An example is {3A 3H 2A 4H} of Figure 9(b). The $LB = 0$, $UB = 61$, $R = 5$, $PS = 12$, and bound at 25. The *LB* alignment operation splits the pattern to {1A} | {2A 3H 2A 4H}. The PCH1 is {3A 3H 2A 4H} with $LB = 1$, $UB = 2 * 12 + 1 = 25$, $R = 2$, $IR = 24$, $PS = 12$. The ECS1={1A} with $LB = 24$, $UB = 26$, $R = 1$, $PS = 1$. The ECS2 is {2A 3H 2A 4H}. The PCH2 is skewed to PCH2'={2A 3H 2A 4H 1A}, as depicted in Figure 9(c). The PCH2' has $LB = 25$, $UB = 50$, $R = 2$, $PS = 12$. The ECS2 has $LB = 49$, $UB = 61$, $PS = 11$, and $R = 1$.

UB alignment. As the operation is applied after *LB* alignment, the LBs are already aligned. Since the *PS* of both patterns is the same, the position of the new aligned *UB* (i.e., the smaller *UB*) is always a factor of the *PS*. Hence, the pattern is not modified. The initial PCH pattern entry is split into two new pattern entries describing the left and the right section of the new bound without traversing the patterns. Hence, the complexity is O(1). The *LB* of the left pattern is the *LB*(*PCH*), the *UB* is the new *UB*, and the repetition factor is $\frac{IR(PCH1)}{PS}$. The remaining pattern has *LB* equal to the bound-1, *UB* equal to the *UB*(*PCH*), and the repetition factor is $\frac{IR(PCH2)}{PS}$. Both new patterns can be a PCH1 ($R > 1$) or an ECS1 ($R = 1$). The ST is not modified, maintaining near-optimality.

7.2.6. PS Modification Operation. When the LCM of the pattern sizes is acceptable (i.e., has a low value avoiding enumeration), the pattern entries are modified to have size equal to the LCM. The new size is achieved by unrolling each pattern by the required Unrolling Factor (*UF*), which is quite low because of the acceptable LCM. The UF for a pattern PCH derives from $UF(PCH) = \frac{LCM}{PS(PCH)}$, thus the complexity is O(1). The pattern is modified by inserting *UF*(*PCH*) times the initial pattern and updating the R to $\frac{R(PCH)}{UF(PCH)}$. The segments of the patterns are not modified, and thus near-optimality is not affected. For instance, the PCH1 is {2A 2H} with $PS = 4$, $R = 30$, and PCH2 is

{2A 1H} with $PS = 3$ and $R = 20$. The LCM is 12. The modified patterns are PCH1= {2A 2H 2A 2H 2A 2H} with $PS = 12$, $R = 10$ and PCH2={2A 1H 2A 1H 2A 1H 2A 1H} with $PS = 12$, $R = 10$.

7.2.7. Subpattern Search Operation. The operation is applied between a pattern with small PS and a pattern with large PS with nonacceptable LCM. If a primitive operation is fully applied, it requires a high UF leading to results close to enumeration. To maintain scalability, the subpattern search operation is proposed, which applies the operation in sections. It searches for repetitive subpatterns in the complete result of the operation, but without computing the complete operation result with unacceptable size. The initial pattern entries are removed, and the new pattern entries are inserted, which describe the subpatterns. The operation is applied in iterations where a subpattern is explored. The subpatterns are described by the segments where the result is not defined upfront. For instance, for the OR operation, the result may not be A only in the positions where both segments have ST=H. Hence, it is sufficient to search for subpatterns with segments of H type. We present the OR operation, but a similar reasoning holds for the AND. In each iteration, the small pattern and a section of the large pattern with size equal to the small pattern are explored. The exploration is performed in a finite set of positions. Variable x defines the starting point of the positions—that is, the start of the large pattern section. To define the complete section, we start from the x position and add segments from the large pattern until the section size is equal to the small PS. Then, the OR operation is applied between the small pattern and the section. If the subpattern contains H, the x position and the subpattern are stored. If the subpattern already exists, only the x position is stored and the R is increased. The process has exploration time O(nlog(n)), as for each subpattern, the previous subpatterns are searched to find if the subpattern exists. Then, the repetition period of the subpatterns is computed, which affects the near-optimality. If the subpattern is not repetitive or with a low R, it can be assumed as accesses. The gain of nonrepetitive patterns cannot be compensated with the significant overhead in the address generation. In this way, we control the introduced near-optimalities. If the subpattern starting positions are regular, the subpattern is maintained with $LB = min(x)$ and $UB = R * PS$. Otherwise, the subpattern is split to smaller subpatterns and regularity is explored. The process is repeated few times—that is, from small $PS(sub - pattern)$ up to a PS equal to 2 (i.e., O(n)). Hence, the operation complexity is O(nlog(n)).

7.2.8. Decomposition to Small Patterns Operation. This operation is similar to the subpattern search operation but applied for patterns with large PS. The operation is applied in iterations with complexity is O(nlog(n)). In each iteration, the operation is applied between the medium pattern and a section of the large pattern of medium size. The positions that are searched are factors of the medium PS. If the resulting pattern contains H, the start position and the resulting pattern (small pattern) are stored. If the small pattern has been already identified, the new start position is stored and the R is increased. The repetition period is also computed. If the small pattern is not repetitive (or has a low R), it can be assumed as virtual accesses due to address generation overhead. These cases are few in number, due to low repetition factor. In this way, the methodology controls the near-optimalities.

7.2.9. Regularity Small Search Operation. This operation is applied when the LCM is not acceptable for a small and a large pattern and the PS of the small pattern is a factor of the PS of the large pattern. The potential repetition of smaller patterns in the result is explored. The operation is applied in pattern sections in a similar way to the postprocessing repetition search, which, however, is applied in pattern segments. Hence, the complexity is O(nlog(n)). The potential repetitive pattern is computed by a

partial OR operation between the first small pattern and a section of the large pattern, with size equal to the small *PS*. The next candidate to be searched is computed by a partial OR operation between the second small pattern and the corresponding large pattern section. If the result is equal to the first result, the pattern is repetitive. The next iteration searches the remaining sections. Otherwise, the potential repetitive pattern is increased by one small pattern (until half the size of the large pattern) and the process is repeated. The scalability is maintained, as the operation is applied in a few pattern sizes (*PScnt*) each time and in a limited set of specific positions in the large pattern (*POScnt*). When scaling up the size, *PScnt* and *POScnt* will not rise proportionally in practical cases.

7.2.10. Regularity Medium Search Operation. The operation is similar to the regularity small search operation, but it is applied when both patterns are large and uses the medium pattern in the position of the small pattern.

8. FINAL SEQUENCE OF PATTERNS AND ARRAY WINDOW

This section describes how the proposed representation is used to compute the array window size for the problem under study in Section 4. Each primitive condition and access statement is described by a pattern entry. Pattern operations over pattern entries are applied to combine them in space and create a final sequence of pattern entries. When several access statements exists, the relevant pattern entries are combined by applying the OR operation. The array window derives from the summation of the lengths of the segments with A behavior in the final sequence of patterns entries. To compute the final sequence of patterns, the operations are applied on a sorted list of pattern entries in increasing LBs and in increasing IR, as second criteria. The first and the second pattern entries are selected, and their parameters are compared to decide in which pattern combination case they belong. Initially, the *PS* of the patterns are compared. When the patterns have different *PS*, the pattern with the larger PS is defined (max pattern). The Greatest Common Divisor (GCD) and the LCM of the pattern sizes are computed. If the LCM has a value that is lower than the acceptable LCM threshold, the acceptable LCM case is active and the *PS* modification operation is applied over the pattern entries. If the LCM is not acceptable, the *PS* of both patterns is compared with the *PS* threshold to distinguish between the Large *PS* and the Small *PS* cases. The value of the GCD decides between the Divisible *PS* case (GCD$>$1) or the Indivisible *PS* case (GCD=1). When the patterns have the same *PS*, potential overlapping of the patterns is explored. When the sum of the *LB* and the *IR* of the PCH1 is larger than the *LB* of the PCH2, the overlapping case is active. To distinguish the different overlapping cases, the sum of the *LB* and *IR* of the two patterns are compared. If they are different, the patterns are not aligned. The *LBs* and the *UBs* are respectively compared to define where the misalignment occurs. In the *UB* alignment case, the repetition factor is compared to define the larger pattern. When the sums of the *LB* and *IR* of the two patterns are equal, the fully aligned case is active. Based on the type of the primitive operation, the OR or AND case is selected. When the sum of the *LB* and *IR* of the PCH1 is equal to the *LB* of the PCH2, the active case is the nonoverlapping sequential patterns. When the sum of the *LB* and *IR* of the PCH1 is less than the *LB* of the PCH2, the active case is the nonoverlapping nonsequential patterns. The corresponding operations are performed in the way described in Section 7.2. The resulting pattern is stored, and the initial patterns are removed from the sorted list. The new patterns created during the pattern operation are also sorted in the list. The process is repeated until the list has a sequence of nonoverlapping pattern entries.

After the computation of the final sequence, it is used to find the number of concurrent alive elements. For the nonoverlapping intrasignal in-place case of one dimension, each

segment in the pattern sequence with a ST equal to A has elements that are accessed and thus are required to be stored. The segments with H behavior describe zero resource requirements. The repetition factor describes the times each PCH pattern is applied in the iteration space. The array window is given by Equation 1.

$$Size_{\mathrm{A}} = \sum_{j=0}^{NumPCH\ Segments(j)} \sum_{i=0} SIR(i)_{(PT==\mathrm{A})} * R + \sum_{j=0}^{NumECH\ Segments} \sum_{i=0} SIR(i)_{(PT==\mathrm{A})} \quad (1)$$

9. EVALUATION

This section uses a case study to demonstrate the proposed representation and presents evaluation results for the case study and for several benchmarks.

9.1. Demonstration Case Study

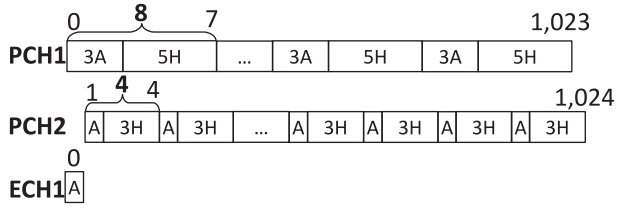
This section demonstrates how the proposed pattern representation is applied to find the sequence of pattern entries. The application code consists of two for nested loops and three manifest conditions over the iterator I, as depicted in Figure 10. For the PCH conditions $(I \geq 8K)\&\&(I \leq 8K+2)\&\&(I < 1024)$, the LB of loop I is defined by $LB = max(-1, 8*(LB_K+1)-1) = max(-1,-1) = -1$, and the UB is defined by $UB = min(1{,}024, 8*(UB_K-1)+8 = min(1{,}024, 8(256-1)+8) = 1{,}024$. The PCH condition $i == 4k+1$ has $LB = max(-1, 4*(LB_k+1)+1-1) = max(-1,0) = 0$. The UB is defined by $UB = min(1{,}025, (LB_k+1)+4*(UB_k-1)+4 = min(1{,}025, 1+4(256-1)+4) = 1{,}025$. Figure 10(a) schematically depicts the patterns of the three conditions. The OR primitive operation is applied between the patterns. The result of merging PCH1 and PCH2 is merged with ECS1. The PCH1 and PCH2 have different PS—that is, $PS(PCH1) = 8$ and $PS(PCH2) = 4$, with an acceptable LCM (i.e., 8). The pattern PCH2 is modified to PCH2' by applying the PS modification operation, as depicted in Figure 10(b). The PCH1 and new PCH2 patterns have both the LB and the UB not aligned. First, the LB alignment operation is applied. The result is depicted in Figure 10(c). The PCH1 has been divided into an ECS2, an ECS3, and a new PCH1 (PCH1'). The UB of the PCH2' is aligned in order to create two fully aligned patterns. The result is depicted in Figure 10(d), where UB of PCH2' has been reduced to the $UB(PCH1')$ and a new ECS4 has been inserted. The PCH1 and PCH2 are fully aligned and the OR operation is applied in Figure 10(e). The ECS conditions are also merged to compose the final sequence of patterns (Figure 10(f)). The array window is given by the A in the final sequence—that is, $1 + 1{,}024/8 * (2+1+1) + 3$.

9.2. Results

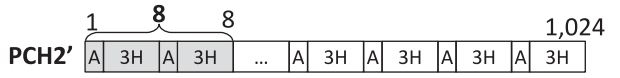
To evaluate the performance of the proposed methodology, we compare it with approaches that are applicable in both regular and irregular iteration spaces—that is, enumerative, symbolic, and approximation approaches. The enumerative approach uses explicitly each access to compute the array window by consistently adding the accesses taking into account all of the read statements. Hence, the enumerative approach has optimal results. The symbolic approaches derive from polytope approaches using barvinok/polylib tool [Verdoolaege 2013] using linear equations to describe the space. An approximation approach to estimate an upper storage size bound in ISH cases derives from approximating the H of the patterns with A (i.e., solidifying the exploration window), which is similar to the approximation of nonuniform access described in Ramanujam et al. [2001]. The upper bound is given by $Size = max(UB_x) - min(LB_x) + 1$, where $LB_x = f_x(i = LB_i, j = LB_j, \ldots)$, $UB_x = f_x(i = UB_i, j = UB_j, \ldots)$, and f_x is index expression of each access. This approach mimics an approximation applied by the

```
For (I=0; I<1,025; I++)
  For (K=0; K<256; K++)
    If ( (I≥8K) && (I≤8K+2) && (I<1024) || (I<1) || (I==4K+1) )
      ...=A[I]
```

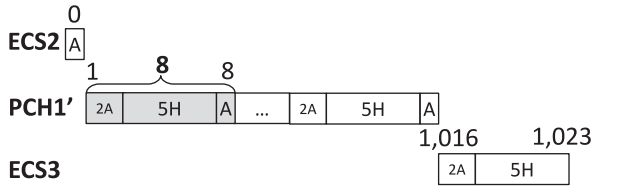
(a)



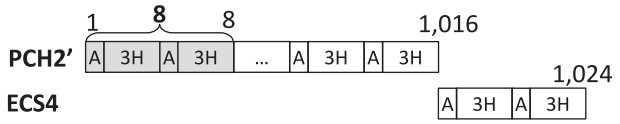
(b)



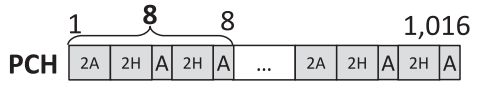
(c)



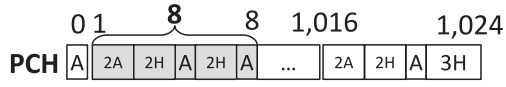
(d)



(e)



(f)



(g)

Fig. 10. Computing storage requirements: application code (a); initial patterns of primitive conditions (b); new PCH2 after PS modification operation (c); new PCH1, ECS2, and ECS3 after LB alignment operation (d); new PCH2 and ECS4 after UB alignment operation (e); result pattern after fully aligned OR operation (f); and final sequence describing the storage requirements (g).

Table II. Comparison Results: Demonstration Case Study, PolyBench(1) and MiBench(2)

Alg.: Array (init.bounds)	Bound Inc.Factor	Near-Optimal Size (Elem.)	Near-Optimal Execution Time (ms) Proposed	Polytopes	Enumerative	Approx. Size (Elem.)
Demonstration case: A (64)	1	32	0.799	50	0.487	507
	3	128	0.804	56	1.694	2,043
	5	512	0.798	56	6.308	8,187
	7	2,048	0.810	59	25.776	32,763
	9	8,192	0.793	60	97.852	131,067
	11	32,768	0.796	58	395.624	524,283
	13	131,072	0.793	59	1,629.237	2,097,147
Correlation: data (1) (32)	1	0.25M	0.343	29	2,886.159	0.25M
	2	1M	0.340	30	11,732.231	1M
	3	4M	0.337	30	48,614.943	4M
	4	16M	0.347	29	188,547.256	16M
	5	64M	0.333	30	780,152.034	64M
	6	256M	0.345	29	-	256M
Jacobi-1D: A (1) (500)	1	1,004	0.674	69	5.428	1,004
	2	2,004	0.660	68	9.998	2,004
	3	4,004	0.676	68	21.646	4,004
	4	8,004	0.664	70	40.701	8,004
	5	16,004	0.656	68	78.056	16,004
	6	32,004	0.658	69	159.161	32,004
Blowfish Decode/Encode: p (2) (2,048)	1	256	0.146	30	6.899	2,040
	7	1,792	0.149	31	45.805	14,328
	13	3,328	0.148	29	86.228	26,616
	19	4,864	0.149	30	120.938	38,904

-:Memory Error produced during simulation.

symbolic/polyhedral approaches, which computes the size by creating the convex hull in the iteration space between the WR and the RD of the elements. For instance, in Figure 1(b) the polyhedral approaches approximate the 2×2 and 1×2 holes by accesses to have a convex hull 11×5. We implemented the proposed representation and the enumerative approach for the nonoverlapping intrasignal in-place optimization step into a Python framework. Code parsing is manually applied in the relevant part of the original code (i.e., the condition statements, the access statements and the loop structure) to derive the initial primitive patterns. The framework takes as input the patterns with the lower and upper bounds of the loops per dimension and creates the patterns entries. The framework sorts the patterns, decides which pattern operations would be applied, and applies them to create the final pattern sequence. The array window is computed based on the different nonoverlapping intrasignal in-place cases. We performed experiments for a set of arrays from benchmarks of the PolyBench [Pouchet et al. 2012], which describe regular spaces, and the MiBench [Guthaus et al. 2001], selected to have ISH to evaluate both the exploration time and the quality of the results in regular and irregular iteration spaces. For each benchmark, we explore different loop bounds by applying a factor in order to increase the number of accesses exploring scalability. The results for the demonstration case study, the PolyBench and the MiBench, are depicted in Table II. The results for MediaBench are depicted in Table III. From the experimental results, it is shown that the proposed methodology achieves optimal results. Only few parameters are changed (i.e., R, UB and IR), not affecting the exploration time, which remains stable as the loop bounds are increased. The exploration times for the proposed methodology are quite close to each other for

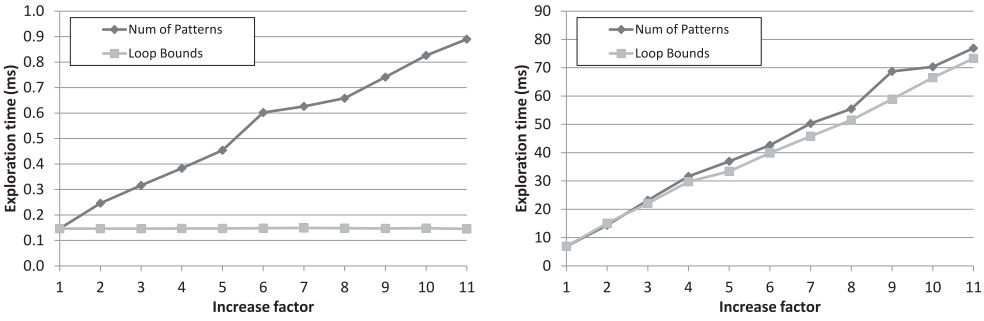
Table III. Comparison Results for MediaBench

Alg.: Array (init.bounds)	Bound Inc.Factor	Near-Optimal				Approx.
			Execution Time (ms)			
		Size (Elem.)	Proposed	Polytopes	Enumerative	Size (Elem.)
Jpeg-Decode&Quant.: DCTblock (256)	1	7,680	0.148	29	34.667	10,239
	5	38,400	0.149	29	162.533	51,199
	9	69,120	0.147	30	281.893	92,159
	13	99,840	0.145	29	413.799	133,119
Epic: image (240)	1	4,257	0.527	125	101.116	11,520
	3	34,609	0.524	127	3,888.289	522,016
	5	138,673	0.529	125	64,824.346	7,986,208
Mpeg: curr_frame (64×32)	1	34,833	1.801	84	852.955	65,537
	2	69,649	1.840	84	1,703.624	131,073
	3	139,281	1.806	86	3,317.909	262,145
	4	278,545	1.813	87	7,167.751	524,289
Motion estimation-full pel: p1/p2 (32)	1	3,576	0.147	31	51.922	12,759
	2	11,512	0.146	29	232.104	65,991
	3	39,672	0.147	30	1,399.778	389,031
	4	145,144	0.147	28	9,093.884	2,589,543
	5	552,696	0.146	32	66,325.246	18,713,319
	6	2,154,232	0.147	29	—	141,892,071
Motion estimation-half pel: p1/p2 (272)	1	40,800	0.149	58	2,181.419	657,152
	2	79,200	0.151	58	8,034	2,492,160
	3	156,000	0.149	59	34,129.439	9,701,120
	4	309,600	0.149	60	133,794.569	38,274,816
	5	616,800	0.148	57	—	152,045,312
Motion estimation-half pel: p1a (272)	1	27,200	0.157	92	2,239.512	657,183
	2	52,800	0.154	89	8,325.536	2,492,191
	3	104,000	0.159	94	36,410.136	9,701,151
	4	206,400	0.156	93	129,134.210	38,274,847
	5	411,200	0.165	90	—	152,045,343
Pgp-outdec: p (128)	1	4,096	0.149	32	45.23	6,144
	6	131,072	0.149	31	683.534	196,608
	11	4,192,304	0.146	29	21,838.798	6,291,456
	16	134,217,728	0.150	32	—	201,326,592

-:Memory Error produced during simulation.

all benchmarks, which is a very promising indication for the limited complexity of our approach. The cases where the time is slightly increased is when more operations are required—for example, increased number of patterns that are misaligned (benchmarks demonstration case study, epic, mpeg). The exploration time of enumerative approach is highly coupled with the number of accesses. The Symbolic/polytope approaches use a set of linear equations that is not modified with an increase over the loop bounds. We observe a gain in exploration time of two orders of magnitude compared with the polytope approach. The approximation leads to high quality loss depending on the number of holes considered as accesses.

Figure 11(a) experimentally describes the exploration time of the proposed methodology relative to (1) an increase factor in the loop bounds (dark gray line) and (2) an increase factor in the number of patterns (i.e., access or condition statements) in the kernel, for benchmark Blowfish-decode/encode. The corresponding exploration times of the enumerative approach are depicted in Figure 11(b). It is verified that the exploration time of the enumerative approach is highly increased with the loops bounds,



(a) Proposed methodology (b) Enumerative approach

Fig. 11. Exploration time comparison (1) when the loop bounds are increased and (2) when the number of patterns in one iteration of the application are increased.

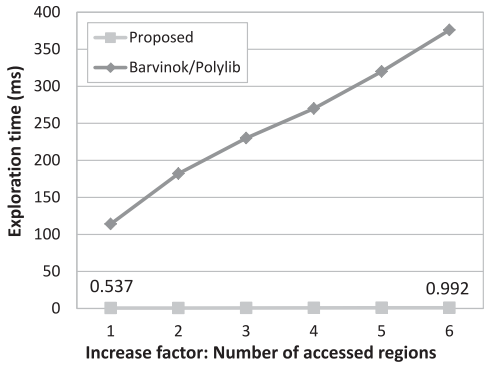


Fig. 12. Exploration time comparison of proposed methodology and barvinok/polylib when the number of accessed regions in one iteration of the application are increased.

whereas the exploration time of the proposed approach remains stable, as the pattern are not modified. When the number of patterns is increased, the exploration time of the enumerative approach is significantly increased, whereas the exploration time of the proposed technique is lower by two orders of magnitude. In realistic applications, the number of patterns is not significantly high, as it is defined by the number of condition and access statements and the pattern operations control the number of new inserted patterns. We also compare the times of the proposed methodology and the symbolic/polytope approaches, as depicted in Figure 12. When the number of accessed regions is increased, the proposed methodology has at least a gain of two orders of magnitude compared to the polytope approaches. As the symbolic approaches do not provide mechanism to trade off between the exploration time and the optimality of the solutions, they cannot control the exploration time of their approach when the number of required linear inequalities to describe the space is highly increased.

10. CONCLUSIONS

This work presents a near-optimal and scalable representation of memory accesses that is applicable for iteration spaces with regular and irregular holes. The proposed methodology (1) introduces the patterns to describe the condition and the access statements, (2) defines the pattern combination cases and presents scalable and near-optimal operations, and (3) applies the patterns and the operations to compute

the maximum number of concurrent alive elements. The results demonstrated that the proposed methodology achieves near-optimal results in low time. The proposed memory accesses representation is useful for the back end of compilers to explore optimizations to reduce the memory size and the transfers in the memory hierarchy and for the memory data layout.

REFERENCES

D. Atienza, J. M. Mendias, S. Mamagkakis, D. Soudris, and F. Catthoor. 2006. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM TODAES* 11, 2, 465–489.

F. Balasa, F. Franssen, F. Catthoor, and H. De Man. 1994. Transformation of nested loops with modulo indexing to affine recurrences. *Let. Parallel Process.* 4, 271–280.

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

A. Bartzas, S. Mamagkakis, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris, and A. Thanailakis. 2006. Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications. In *Proceedings of DATA*. ACM, 740–745.

F. Catthoor. 1999. Energy-delay efficient data storage and transfer architectures and methodologies: Current solutions and remaining problems. *J. VLSI Signal Process.* 21, 3, 219–231.

F. Catthoor, E. De Greef, and S. Suytack. 1998a. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer.

F. Catthoor, S. Wuytack, E. De Greef, F. Franssen, L. Nachtergaele, and H. De Man. 1998b. System-level transformations for low power data transfer and storage. In *Proceedings of Low-Power CMOS Design*. IEEE, 609–618.

D. Cho, I. Issenin, N. Dutt, J. W. Yoon, and Y. Paek. 2007. Software controlled memory layout reorganization for irregular array access patterns. In *Proceedings of CASES*. ACM, 179–188.

P. Clauss and B. Meister. 2000. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News* 28, 1, 11–19.

J. Cong, W. Jiang, B. Liu, and Y. Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM TODAES* 16, 2, 15:1–15:25.

B. Creusillet and F. Irigoin. 1996. *Exact vs. Approximate Array Region Analyses*.

K. Danckaert, F. Catthoor, and H. De Man. 1999. Platform independent data transfer & storage exploration illustrated on parallel cavity detection algorithm. In *Proceedings of PDPTA*. 1669–1675.

A. Darte, R. Schreiber, and G. Villard. 2005. Lattice-based memory allocation. *IEEE Trans. Comput.* 54, 10, 1242–1257.

J. Dongarra, O. Brewer, S. Fineberg, and J. A. Kohl. 1990. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *J. Parallel Distrib. Comput.* 9, 2, 185–202.

F. H. M. Franssen, F. Balasa, M. F. X. B. van Swaaij, F. V. M. Catthoor, and H. J. De Man. 1993. Modeling multidimensional data and control flow. *Tran. VLSI* 1, 3, 319–327.

S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. De Bosschere. 2009. System scenario based design of dynamic embedded systems. *ACM TODAES* 14, 3, 1–45.

A. Grösslinger. 2009. Precise management of scratchpad memories for localising array accesses in scientific codes. In *Proceedings of the International Conference on Compiler Construction*. Springer, Berlin, 236–250.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the International Workshop on Workload Characterization*. IEEE, Washington, DC, 3–14.

B. Jang, D. Schaa, P. Mistry, and D. Kaeli. 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *TPDS* 22, 105–118.

T. Janjusic, K. M. Kavi, and B. Potter. 2011. Gleipnir: A memory analysis tool. In *Proceedings of the International Conference on Computational Science*. Springer, Netherlands, 2058–2067.

P. K. Jha and N. D. Dutt. 1997. Library mapping for memories. In *Proceedings of EDAC*. IEEE, 288.

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

P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. 2003. Data dependency size estimation for use in memory optimization. *TCAD* 22, 908–921.

P. G. Kjeldsberg, F. Catthoor, and E. J. Aas. 2004. Storage requirement estimation for optimized design of data intensive applications. *ACM TODAES* 9, 2, 133–158.

A. Kritikakou, F. Catthoor, V. Kelefouras, and C. Goutis. 2013a. Near-optimal & scalable intra-signal in-place optimization for non-overlapping & irregular access schemes. *ACM TODAES* 19, 1, Article 4.

A. Kritikakou, F. Catthoor, V. Kelefouras, and C. Goutis. 2013b. Scalable & near-optimal array size under overlapping & irregular accesses. *IEEE Trans. Comput.* Under revision.

C. Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. MediaBench: A tool for evaluating & synthesizing multimedia & communications systems. In *Proceedings of MICRO*. IEEE, 330–335.

S.-T. Leung and J. Zahorjan. 1995. *Optimizing Data Locality by Array Restructuring*. Technical Report.

P. E. R. Lippens, J. L. Van Meerbergen, W. F. J. Verhaegh, and A. Van Der Werf. 1993. Allocation of multiport memories for hierarchical data stream. In *Proceedings of CAD*. IEEE, 728–735.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. f. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.* 40, 6, 190–200.

M. Martonosi, A. Gupta, and T. Anderson. 1992. MemSpy: Analyzing memory system bottlenecks in programs. *SIGMETRICS Perform. Eval. Rev.* 20, 1, 1–12.

D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. 1993. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2–15.

L. Nachtergaele, I. Bolsens, and H. De Man. 1992. A specification and simulation front-end for hardware synthesis of digital signal processing applications. *Int. J. Comp. Simulation* 2, 213–229.

O. Ozturk, M. Kandemir, and G. Chen. 2008. Access pattern-based code compression for memory-constrained systems. *ACM Trans. Des. Autom. Electron. Syst.* 13, 4, 60:1–60:30.

Y. Paek, J. Hoeflinger, and D. Padua. 2002. Efficient and precise array access analysis. *ACM TOPLAS* 24, 1, 65–109.

K. V. Palem, R. M. Rabbah, V. J. Mooney, P. Korkmaz, and K. Puttaswamy. 2002. Design space optimization of embedded memory systems via data remapping. In *Proceedings of LCTES*. ACM, 28–37.

P. R. Panda, N. D. Dutt, and A. Nicolau. 1999. Local memory exploration and optimization in embedded systems. *TCAD* 18, 1, 3–13.

P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. 2001. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.* 6, 2, 149–206.

L. N. Pouchet, J. Cavazos, and S. Grauer-Gray. 2012. *PolyBench/C:The Polyhedral Benchmark Suite*. Retrieved from http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

J. Ramanujam, J. Hong, M. Kandemir, and A. Narayan. 2001. Reducing memory requirements of nested loops for embedded systems. In *Proceedings of DAC*. ACM, 359–364.

S. Rubin, R. Bodík, and T. Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. *SIGPLAN Not.* 37, 1, 140–153.

R. Seghir, V. Loechner, and B. Meister. 2012. Integer affine transformations of parametric polytopes and applications to loop nest optimization. *ACM TACO* 9, 2, 8:1–8:27.

Z. Shen, Z. Li, and P.-C. Yew. 1990. An empirical study of Fortran programs for parallelizing compilers. *Trans. Parallel Distrib. Syst.* 1, 356–364.

B. So, M. W. Hall, and H. E. Ziegler. 2004. Custom data layout for memory parallelism. In *Proceedings of ISSGO*. IEEE, 291.

S. Verdoolaege. 2013. Barvinok. Retrieved from http://barvinok.gforge.inria.fr/.

M. Swaaij, F. Franssen, F. Catthoor, and H. De Man. 1992. Automating high level control flow transformations for DSP memory management. In *Proceedings of the Workshop on VLSI Signal Processing*. IEEE, 397–406.

T. Van Achteren and R. Lauwereins. 2000. Systematic data reuse exploration methodology for irregular access patterns. In *Proceedings of the International Symposium on System Synthesis*. IEEE, 115–121.

S. Wuytack, J.-P. Diguet, F. Catthoor, and H. De Man. 1997. Formalized methodology for data reuse exploration in hierarchical memory mappings. In *Proceedings of ISLPED*. IEEE, 30–35.

S. Wuytack, J.-P. Diguet, F. Catthoor, and H. De Man. 1998. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. *TVLSI* 6, 4, 529–537.

Received November 2013; accepted January 2014