Chapter 4

Polyhedral Techniques for Parametric Memory Requirement Estimation

Philippe Clauss
University of Strasbourg, France

Diego Garbervetsky
Universidad de Buenos Aires, Argentina

Vincent Loechner
University of Strasbourg, France

Sven Verdoolaege
INRIA Saclay, France

Contents

4.1 Introduction

Memory requirement estimation is an important issue in the development of embedded systems, since memory directly influences performance, cost, and power consumption. It is therefore crucial to have tools that automatically compute accurate estimates of the memory requirements of programs to better control the development process and avoid some catastrophic execution exceptions.

Compute-intensive applications often spend most of their execution time in nested loops. The polyhedral model provides a powerful abstraction to reason about analyses and transformations on such loop nests by viewing an instance, or iteration, of each statement as an integer point in a polyhedron. From such a representation and a precise characterization of inter and intra-statement dependences, it is possible to analyze loop nests statically in a completely mathematical setting relying on machinery from linear algebra, integer linear programming and polynomial algebra. The analyses made on integer points in polyhedra correspond to effective quantitative characteristics of the original code such as the variable liveness or the amount of consumed memory.

The polyhedral model is applicable to loop nests in which the data access functions and loop bounds are affine combinations of the enclosing loop indices and parameters. While a precise characterization of data dependences is feasible for programs with static control structures and affine references and loop bounds, codes with non-affine array access functions or code with dynamic control can also be handled, but either with conservative assumptions on some dependences, or with additional knowledge coming from developers, or from advanced profiling/modeling techniques.

In this chapter, we present several techniques using the polyhedral model and providing automatic estimations of memory requirements in nested-loops programs. In the next section, useful background on the polyhedral model is provided. Section 4.3 relates the computation of the exact number of touched memory locations during the execution of a loop nest with the general issue

of computing the number of integer points in the affine transformation of a polyhedron. Section 4.4 considers the computation of the maximum memory amount required during the execution of loop nests, while using temporary variables or dynamically allocated memory. The proposed technique is related to the general issue of maximizing a parametric multivariate polynomial defined over a polyhedron. Finally, conclusions are given in Section 4.5.

4.2 The Polyhedral Model of Loop Nests

4.2.1 Loop nests and polyhedra

In this chapter, we will consider loop nests with increments of one, and the statements not modifying the loop indices, nor exiting prematurely the loop. The loop bounds of each loop are affine functions of the outer loop indices and of some parameters (variables with unknown values, but that can be affinely constrained to each other).

Example 4.1 Consider for example the following loop nest of depth two:

```
for (i = 0; i < n; i++)
{
   /* S1(i) */
   for (j = i; j < i+n; j++)
   {
      /* S2(i,j) */
   }
   /* S3(i) */
}
```

where S1, S2 and S3 are three statements, containing no instructions `break`, `continue` and `exit`, using but not modifying `i` and `j`.

The i-loop bounds $min_i = 0$ and $max_i = n$ are affine functions of a parameter n, and the j-loop bounds $min_j = i$ and $max_j = i + n$ are affine functions of the parameter and the outer loop index i.

Definition 4.1 (iteration domain) *An iteration domain is associated to each statement: it is the set of iterations of the loop nest enclosing the statement.*

The following definitions introduce some geometric concepts. We start with the concept of a parametric polyhedron, already briefly touched upon in Chapter **??**.

Au: Please supply chap. no.

Definition 4.2 (parametric polyhedron) *A parametric polyhedron is the intersection of a finite number of affine half-spaces, defined as a set of affine inequalities on the indices and parameters:*

$$\mathcal{P}(\mathbf{p}) = \{\mathbf{z} \mid A\mathbf{z} + B\mathbf{p} + \mathbf{c} \geq 0\}$$

where z is the vector of indices, p is a vector of parameters, A and B are matrices, c is a vector.

The iteration domain of each statement of an affine loop nest as defined so far in this section is the set of integer points contained in a parametric polyhedron.

Example 4.1 (continued) The iteration domain of statement S1 is:

$$\mathcal{D}_{S1}(n) = \{i \in \mathbb{Z} \mid i \geq 0 \wedge i < n\}$$

The iteration domain of statement S2 is:

$$\mathcal{D}_{S2}(n) = \{(i,j) \in \mathbb{Z}^2 \mid 0 \leq i < n \wedge i \leq j < i + n\}$$

Definition 4.3 (polyhedral integer set) *A* polyhedral integer set *is a subset of $\mathbb{Z}^d$ defined as a set of integer vectors constrained by affine inequalities on the indices, on the parameters, and on integer valued existentially quantified variables.*

The class of non-parametric polyhedral integer sets is the same as that of the linearly bounded lattices used in Chapter **??**.

Au: Please supply.

Property 4.1 A *polyhedral integer set* geometrically is a finite union of parametric polytopes intersected with integer lattices [1,2].

The set of references accessed through an affine array in a loop nest is a polyhedral integer set.

Example 4.2 Consider the following loop nest:

```
for (i = 0; i < n; i++)
{
   for (j = i; j < i+n; j++)
   {
      A[2*j-i] = i+j ; /* S2(i,j) */
   }
}
```

The elements of array A being accessed by this loop nest in statement S2 is the following polyhedral integer set, where the affine access reference is denoted by x, and the loop iterators i and j are existentially quantified variables:

$$\{x \in \mathbb{Z} \mid \exists (i,j) \in \mathbb{Z}^2 \text{ such that } 0 \leq i < n \wedge i \leq j < i+n \wedge x = 2j - i\}$$

The iteration domain of a statement in a loop nest with *quasi-affine bounds* (using floor, ceil, modulo, or fractional part of division) is a polyhedral integer set. Indeed, these functions can be transformed using an extra existentially quantified variable.

Example 4.3 The equality

$$x = \lfloor y/z \rfloor$$

with $x, y, z \in \mathbb{Z}$ is equivalent to:

$$\exists t \in \mathbb{Z} \text{ such that } 0 \leq t < z \wedge zx = y + t$$

The following definition introduces the concept of relation in our context. Usually, a binary relation on a set S is defined as a collection of ordered pairs of elements of S, or equivalently as a subset of the cartesian product $S \times S$. We will define it in a very similar way, but making it depend on a vector of parameters.

Definition 4.4 (polyhedral relation) *A* polyhedral relation *is a function:*

$$\begin{array}{rcl} R : \mathbb{Z}^m & \rightarrow & \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \\ p & \mapsto & \{z_1 \rightarrow z_2 \mid (z_1, z_2) \in \mathcal{D}(p)\} \end{array}$$

where p is the parameters vector and $\mathcal{D}$ is a polyhedral integer set.

Example 4.4 Consider the loop nest of Example 4.2. The access relation of array A in statement S2 is:

$$R(n) = \{(i,j) \rightarrow x \mid 0 \leq i < n \wedge i \leq j < i+n \wedge x = 2j - i\}$$

4.2.2 Data-flow analysis

The data-flow analysis consists in computing which resource access is dependent on which other. A resource is usually a variable or an array in the source code, compiled as a memory location or a register in assembly.

The data-flow graph is a subgraph of the control flow graph of the program (the graph of all paths that might be traversed through a program during its execution), since the origin of the dependence executes before the endpoint of the dependence.

Definition 4.5 (dependence) *Let S1 and S2 be two statement instances of a program. There is a dependence from S1 to S2 if:*

- *S1 is executed before S2;*
- *S1 and S2 access the same resource, and at least one of the statements modifies the resource;*
- *there is no statement executed between S1 and S2 that modifies the resource.*

Notice that there is no dependence if both statements read the same resource: reading the same data twice can be done in no particular order. However, this kind of *input "dependence"* can be used by the compiler to make some optimizations (in particular cache optimizations).

If S1 modifies data being read by S2, it is said to be a *flow (or true) dependence*. If S1 reads data being modified afterwards by S2, it is an *antidependence*. If both S1 and S2 modify the same data, it is an *output dependence*.

The following definition formalizes this notion of dependence between array accesses in loop nests as a quasi-affine polyhedral relation.

Definition 4.6 (dependence relation) *Let S1 and S2 be two statements accessing a given array, executed at iteration z_1 and z_2 of their enclosing loop nests respectively, without any other intermediate access from any other statement. The dependence relation between those two array accesses is a polyhedral relation $z_1 \rightarrow z_2$, where z_1 is the last iteration of statement S1 accessing to the array, included in a polyhedral integer set defined by:*

- *z_1 and z_2 are iterations of the loop nests enclosing S1 and S2 respectively;*
- *z_1 is lexicographically smaller than z_2 if S1 and S2 are in the same loop nest, else the loop nest enclosing S1 is executed before the loop nest enclosing S2: this ensures S1(z_1) executes before S2(z_2), as stated in the first item of Definition 4.5;*
- *$a_1(z_1) = a_2(z_2)$, with a_1 and a_2 the arrays access functions in S1 and S2: the two statements access the same resource (second item of Definition 4.5).*

Notice that the third item of Definition 4.5 is ensured by computing the last iteration z_1 of this polyhedral integer set.

An example of such a computation is given in Section 4.4.1, Example 4.8.

4.2.3 Software tools

4.2.3.1 PipLib

PipLib[1] is a parametric integer linear programming solver library. It finds the lexicographic minimum (or maximum) in the set of integer points belonging to

[1]First release of PIP 1988 - Development: http://piplib.org

a convex polyhedron. It is based on parametric Gomory's cuts and on the parametric dual simplex method. It can be used to solve the dependence system.

4.2.3.2 PolyLib

PolyLib[2] is a library of polyhedral functions, that can manipulate unions of rational polyhedra of any dimension as described in Chapter **??**. It was the first to provide an implementation of the computation of parametric vertices of a parametric polyhedron, and the computation of an Ehrhart polynomial (expressing the number of integer points contained in a parametric polytope, as presented in next Section 4.3), based on an interpolation method.

Au: Please supply.

4.2.3.3 Omega

The Omega+ Library[3] provides all basic operations on polyhedral integer sets and relations. It includes the *Omega Test*, a decision test for the existence of integer solutions to affine constraints, that can be used for dependence analysis. The key transformation implemented in this project is the elimination of integer existentially quantified variables. It also contains a code generation library, to generate loop nests from polyhedral integer sets.

4.2.3.4 PPL: The parma polyhedra library

The Parma Polyhedra Library[4] can manipulate partially open rational polyhedra. It also provides a mixed integer linear programming problem solver using an exact-arithmetic version of the simplex algorithm, and a parametric integer programming solver. It is user friendly, portable, and has a very clean design.

4.2.3.5 The integer set library (isl)

The Integer Set Library[5] manipulates polyhedral integer sets and relations, by exclusively using a constraints based representation. It provides computation of the integer projection, the integer affine hull, the lexicographic minimum using parametric integer programming, and Bernstein expansion.

4.2.3.6 CLooG

CLooG generates code for scanning the elements of a collection of polyhedral integer sets based on an extension of the algorithm for a single parametric polytope presented in Chapter **??**.

Au: Please supply.

[2]First release 1993 - http://icps.u-strasbg.fr/PolyLib

[3]First release of the Omega Project 1992 - http://www.cs.utah.edu/ chunchen/omega/

[4]First release 2001 - http://www.cs.unipr.it/ppl/

[5]First release 2009 - http://freshmeat.net/projects/isl/

4.2.3.7 Barvinok

The barvinok library[6] implements the computation of the number of elements in parametric integer sets, based on Barvinok's counting algorithm. The same algorithm, restricted to non-parametric polytopes, was first implemented in LattE and was later also reimplemented in K2 (see Chapter **??**). The Barvinok distribution also includes an interactive tool called iscc that exposes some of the functionality of isl, Barvinok and CLooG, and that will be used in some of the examples in this chapter.

Au: Please supply.

4.2.3.8 PoCC and PLuTo

The PoCC package (Polyhedral Compiler Collection)[7] a source-to-source iterative compiler, which contains Clan (the Chunky loop analyzer) to extract a polyhedral intermediate representation from the source code; the Chunky analyzer for dependences in loops (Candl) to compute polyhedral dependences; LetSee (Legal transformation Space explorator) for finding different affine multidimensional schedules; the automatic parallelizer and locality optimizer PLuTo[8], for finding a valid tiling and parallel loop nest in the polyhedral model; CLooG (Chunky Loop Generator), to generate loop nests code; PipLib; PolyLib; and FM (Fourier-Motzkin library).

4.3 Counting the Elements in a Polyhedral Set

In many memory requirement estimation problems, the memory requirements fluctuate during the execution of the program. Solving such a problem then involves two steps. In a first step, the memory requirements of the program at any given point during the execution are computed. In the second step, the ultimate memory requirements of the complete program are obtained as a bound on this local memory requirement over the entire execution of the program. The first step is usually a counting problem and these counting problems are the subject of this section. The prototypical example is that of computing the maximal number of live memory elements in a program, where the first step consists of computing the number of live memory elements at any given point during the program execution.

Before we discuss this prototypical example, we first consider the counting problem in a slightly more general setting. Next, we explain how to compute reuse distances and we finish this section with a discussion on the

[6]First release 2003 - http://freshmeat.net/projects/barvinok/

[7]First release 2009 - http://pocc.sourceforge.net/

[8]First release 2008 - http://pluto-compiler.sourceforge.net/

```
1 for (i = 1; i <= n; ++i)
2     for (j = 1; j <= i; ++j)
3         /* S */
```

Figure 4.1 A simple loop nest.

generalized problem of weighted counting. Throughout this section, we will focus on how to *model* counting problems. For details on how to *solve* these counting problems, we refer to the literature: [3] for an overview; [4–6] for non-parametric problems; [7–12] for parametric problems; and [13] for weighted counting problems.

4.3.1 Introduction

As an introduction to counting, we will consider a few simple loop nests and ask ourselves a very simple question: how many times is the body of the loop nest executed?

The first loop nest is shown in Figure 4.1. Let us first assume that `n` has a fixed value, say 5. The iteration domain of the loop nest can be described as

$$\{\,(i,j) \mid 1 \le i \le 5 \wedge 1 \le j \le i\,\}$$

and is shown in Figure 4.2. The number of iterations of the loop nest is equal to the number of elements in this iteration domain, i.e.,

$$\#\{\,(i,j) \mid 1 \le i \le 5 \wedge 1 \le j \le i\,\} = 15,$$

which the reader can easily verify by hand. Usually, however, we are interested in solving such counting problems for arbitrary values of the parameters, i.e., in this case, for arbitrary values of `n`. The answer is then not just a number,

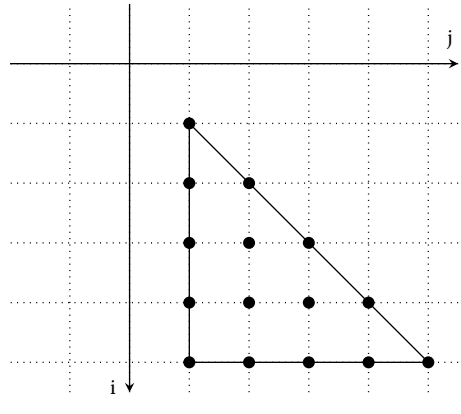

Figure 4.2 The iteration domain of the loop nest in Figure 4.1.

but a formula in terms of the parameters. For the loop nest in Figure 4.1, we obtain

$$\#\{\,(i,j) \mid 1 \le i \le n \wedge 1 \le j \le i\,\} = \frac{n(n+1)}{2}. \tag{4.1}$$

The previous result is indeed a special case of this formula, as can be verified by plugging in the value $n = 5$.

The formula in Equation 4.1 describing the number of iterations of the loop nest in Figure 4.1 is a polynomial in the parameter(s). In general, however, the result of such a counting problem is not representable using a single polynomial. Assume, for example, that the upper bound on the `i`-loop in Figure 4.1 is not simply `n`, but `min(n, m)`, where `m` is a second parameter. The number of iterations can now be described as

$$\#\{\,(i,j) \mid 1 \le i \le n,m \wedge 1 \le j \le i\,\} = \begin{cases} \dfrac{n(n+1)}{2} & \text{if } 1 \le n \le m \\ \dfrac{m(m+1)}{2} & \text{if } 1 \le m \le n. \end{cases} \tag{4.2}$$

Notice that two polynomials are needed to describe the number of iterations, one that is valid when `n` is smaller than `m` and one that is valid in the other case. For this simple example, this result can easily be explained by the fact that the smaller of the two parameters determines the number of iterations in the outer loop and therefore also the total number of iterations. In principle, there is also a third case, where either `n` or `m` is smaller than 1, where the loop is not executed and the number of iterations is zero. We will usually omit such cases and assume that the result of the counting problem is zero for all cases that are not explicitly mentioned. The different cases will be called *cells*.

Let us try the example above in `iscc`. The set $\{\,(i,j) \mid 1 \le i \le m,n \wedge 1 \le j \le i\,\}$ is represented as

```
[m,n] -> { [i,j] : 1 <= i <= m,n and 1 <= j <= i }
```

with the list of parameters separated from the main set description by a `->`. Computing

```
card [m,n] -> { [i,j] : 1 <= i <= m,n and 1 <= j <= i };
```

results in

```
[m, n] -> { (1/2 * n + 1/2 * n^2) : n <= m and n >= 1;
            (1/2 * m + 1/2 * m^2) : m >= 1 and n >= 1 + m }
```

Notice that the second domain has a constraint $m \le n - 1$ instead of $m \le n$. The reason is that we usually prefer to have disjoint cells. This preference will be partly explained in Section 4.3.4 where we discuss incremental counting. In Equation 4.2, the two cells overlap at $n = m$ and on this overlap the two polynomials are equivalent. Here, the overlap has been removed from one of the two cells.

```
1 for (i = 1; i <= n; ++i)
2     for (j = 1; j <= n - 2 * i; ++j)
3         /* S */
```

Figure 4.3 A slightly less simple loop nest.

Besides polynomials and cells, we need one final ingredient to describe the number of elements in a polyhedral set in general. To show the need for this final ingredient, consider the loop nest in Figure 4.3. The iteration domain can be described as

$$\{\,(i,j) \mid 1 \le i \le n \wedge 1 \le j \le n - 2i\,\}.$$

This iteration domain is shown in Figure 4.4 for varying values of `n`. Note that the upper bound on the `i`-loop is slightly misleading, as the `j`-loop is only executed for values of `i` that result in an upper bound that is at least as large as the lower bound, i.e., for $1 \le n - 2i$ or $i \le (n-1)/2$. Also note that the vertices of the iteration domains in Figure 4.4 are not always integral. In fact, they are only integral for even values of n. This nonintegrality results in "jumps" in the number of iterations when going from an odd value of n to an even value of n. The effect of these jumps is that two polynomials are required to describe the number of iterations in the loop nest, one for even n and one for odd n, namely,

$$\begin{cases} -\frac{n}{2} + \frac{n^2}{4} & \text{if } \exists \alpha : n \ge 4 \wedge n = 2\alpha \\ \frac{1}{4} - \frac{n}{2} + \frac{n^2}{4} & \text{if } \exists \alpha : n \ge 3 \wedge n = 2\alpha + 1. \end{cases} \tag{4.3}$$

This pair of polynomials is shown in Figure 4.5. They can be captured by a *single* polynomial expression if we allow this expression to contain floors of

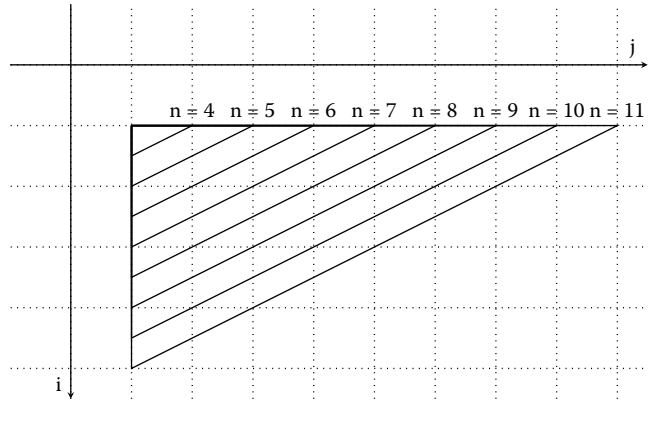

Figure 4.4 Iteration domains of the loop nest in Figure 4.3 for varying values of n.

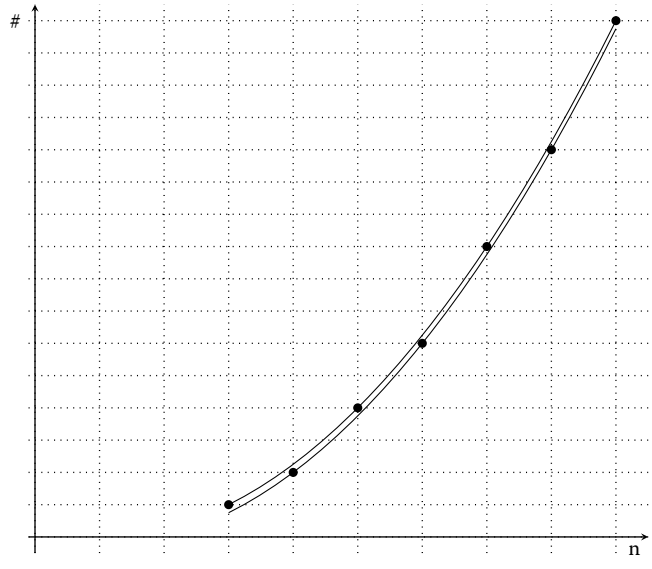

Figure 4.5 The number of elements in the iteration domains in Figure 4.4.

affine expressions. In particular, the function in Equation 4.3 can be described as

$$-\frac{n}{4} + \frac{n^2}{4} - \frac{1}{2}\left\lfloor \frac{n}{2} \right\rfloor \quad \text{if } n \geq 3. \tag{4.4}$$

The two polynomials in Equation 4.3 can easily be recovered from this expression if we note that $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$ for even n, while $\lfloor \frac{n}{2} \rfloor = \frac{n}{2} - \frac{1}{2}$ for odd n. We generally prefer the representation in Equation 4.4 because it is usually more compact than a representation using only polynomials, as in Equation 4.3. This more compact representation is also what `iscc` will give you by default. That is,

```
card [n] -> { [i,j] : 1 <= i <= n and 1 <= j <= n - 2i };
```

results in

```
[n] -> { ((-1/4 * n + 1/4 * n^2) - 1/2 * [(n)/2]) : n >= 3 }
```

We are now ready to describe the counting problem on polyhedral sets and relations. We first define step-polynomials and piecewise step-polynomials.

Definition 4.7 (Step-polynomial) *A* step-polynomial $q(\mathbf{x})$ *in the integer variables* $\mathbf{x}$ *is a rational polynomial expression in greatest integer parts of rational affine expressions in the variables, i.e.,* $q(\mathbf{x}) \in \mathbb{Q}\left[\lfloor \mathbb{Q}[\mathbf{x}]_{\leq 1} \rfloor\right]$.

A small note on the notation: $\mathbb{Q}[\cdot]$ represents polynomial expressions in "$\cdot$" and $\mathbb{Q}[\cdot]_{\leq d}$ represents polynomial expressions in "$\cdot$" of degree at most d. We will usually consider step-polynomials in either both parameters and variables or in only parameters.

Definition 4.8 (Piecewise step-polynomial) *A* piecewise step-polynomial $q(\mathbf{x})$*, with* $\mathbf{x} \in \mathbb{Z}^d$ *consists of a finite set of pairwise disjoint polyhedral sets* $K_i \subseteq \mathbb{Z}^d$*, called* cells*, each with an associated step-polynomial* $q_i(\mathbf{x})$*. The value of the piecewise step-polynomial at* $\mathbf{x}$ *is the value of* $q_i(\mathbf{x})$ *with* K_i *the cell containing* $\mathbf{x}$ *or zero if no cell contains* $\mathbf{x}$*, i.e.,*

$$q(\mathbf{x}) = \begin{cases} q_i(\mathbf{x}) & \text{if } \mathbf{x} \in K_i \\ 0 & \text{otherwise.} \end{cases}$$

The operation of counting the number of elements in a set then simply takes a parametric polyhedral set as input and produces a piecewise step-polynomial in the parameters describing the number of elements in the set.

Operation 4.1 (Number of elements in a set)
Input: a polyhedral set $S : \mathbb{Z}^n \to \mathbb{Z}^d$
Output: a piecewise step-polynomial $q : \mathbb{Z}^n \to \mathbb{Q} : (\mathbf{s}) \mapsto q(\mathbf{s}) = \#S(\mathbf{s})$

Note that the type of q is that of a function that returns a rational number, but the actual function values will always be integers.

There are different ways of defining a counting operation on a relation. One possibility would be to count the total number of pairs of domain and image elements. However, for our applications, it is more convenient to define an operation that counts the number of elements in the image of an arbitrary domain element.

Operation 4.2 (Number of image elements in a relation)
Input: a polyhedral relation $R : \mathbb{Z}^n \to \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}$
Output: a piecewise step-polynomial $q : \mathbb{Z}^n \times \mathbb{Z}^{d_1} \to \mathbb{Q} : (\mathbf{s}, \mathbf{t}) \mapsto q(\mathbf{s}, \mathbf{t}) = \#R(\mathbf{s}, \mathbf{t})$

Note that this operation is equivalent to treating the domain dimensions as extra parameters and then counting the number of elements in the resulting set.

4.3.2 The number of live memory elements

In this chapter, we extend the computation of data storage requirements of Chapter **??** to programs that may involve parameters and that need not necessarily be in single-assignment form. As explained before, this computation involves two steps. We first compute the number of memory elements live at

Au: Please supply.

any given point during the execution and then compute an upper bound on this number over the entire execution of the program, which is then also an upper bound on the minimal memory requirements of the program. The second step involves an approximation and will be explained in Section 4.4. The first step can be performed exactly and is explained in this section.

Recall from Section 4.2.2 that dataflow analysis determines for each read access, the write access that wrote the value being read. The corresponding memory element is then clearly live at execution points between the write and the read. However, for *counting* the number of live elements, the standard dataflow relations are not very practical, because a given memory element may be involved in more than one of these relations at the same time if the result of a write is read multiple times. Counting can be made substantially easier by constructing injective relations that map the statement instance that produces an element to the statement instance that kills it, i.e., where it is used for the last time. Below, we describe two ways of constructing such relations. For simplicity, we assume here that each statement instance produces at most element, so that we can use the statement instance as an identifier for the element being produced.

- One way is to essentially apply dataflow analysis twice. In the first application, we determine for each *write*, what is *next* write to the same memory element. In the second application, we determine for each write, what is the *last* subsequent read to the same memory element, *before* the next write, or, for those writes without a next write, simply the last subsequent read. The resulting relations clearly capture the births and deaths of memory elements. An alternative method of computing the same relation is to first perform standard dataflow analysis, relating each read to the last preceding write and to then compute the last read associated to each write in the resulting relation.
- The second way is to determine for each read, what is the previous write *or read*. In this case, a memory element is considered dead after the first read, but is immediately revived if there is a second read. This resuscitation process then continues until the final read.

Example 4.5 Consider the sequence of writes and reads to a single memory element shown in Figure 4.6. The top row illustrates the first way of computing liveness relations. First, the next write is computed. This results in the relation

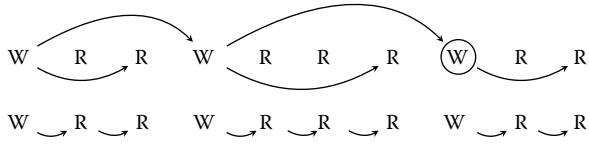

Figure 4.6 Writes and reads of a single memory element and the corresponding liveness relation.

shown above the first row and the encircled last write, which by definition has no next write. Based on this information the last read before the next write is computed and the result is shown below the top row. The second row illustrates the second way of computing liveness relations.

In general, the first way is more expensive, because dataflow analysis is applied twice, but the resulting relations have fewer elements (i.e., fewer arrows in Figure 4.6). Note, though, that this does not necessarily mean that computing the number of elements will be cheaper, as this computation is performed entirely symbolically.

Once a suitable injective relation R has been computed, there are again at least two ways of computing the number of elements live at iteration $\mathbf{i}$.

- The first way is to count all elements in the relation such that the source precedes $\mathbf{i}$ and the sink follows $\mathbf{i}$, i.e.,

$$L(\mathbf{i}) = \#\{(\mathbf{s}, \mathbf{t}) \in R \mid \mathbf{s} \prec \mathbf{i} \preccurlyeq \mathbf{t}\}.$$

- The second way is to count both the sources and the sinks that precede $\mathbf{i}$. The difference is equal to those preceding sources of which the corresponding sink does not precede $\mathbf{i}$, i.e.,

$$L(\mathbf{i}) = \#\{\mathbf{s} \in \operatorname{dom} R \mid \mathbf{s} \prec \mathbf{i}\} - \#\{(\mathbf{s}, \mathbf{t}) \in R \mid \mathbf{t} \prec \mathbf{i}\}.$$

  Note that we cannot use $\mathbf{t} \in \operatorname{ran} R$ in the second counting problem because a given statement instance may kill more than one element.

It is difficult to predict which of these two methods will be more efficient. Since the lexicographical order needs to be linearized, the number of basic relations may grow by a factor of d^2 in the first method (with d the loop nest depth), while it only grows by a factor of up to $2d$ in the second method. On the other hand, basic relations that lie entirely before $\mathbf{i}$ are counted twice in the second method, while they are not considered at all in the first method.

4.3.3 Reuse distances

Modern computer architecture typically have a hierarchy of caches. Each cache stores recently used data. If this data is later reused, an access to a slower cache or the main memory can be avoided. Whenever a new value is stored in the cache, some other value needs to be evicted. There are many policies for deciding which value to evict. One such policy is to evict the least recently used (LRU) value. If the number of distinct data elements accessed between two consecutive accesses to the same data element is smaller than the cache size, then this this data element will still be in the cache, assuming the cache is LRU and fully associative. This number of distinct elements is called the reuse distance and its computation is useful for analyzing and optimizing cache behavior [14].

Pairs of consecutive accesses to the same memory element can be obtained from dataflow analysis (Section 4.2.2), where now no distinction is made between read and write accesses and all accesses are treated as both potential sources and potential sinks. Depending on the desired accuracy of the analysis, the access relations may need to be composed with allocation relations that map array indices to memory elements or even cache lines. The dataflow analysis yields a set of relations $R_{r\to s}$ of consecutive accesses, one for each pair of references r and s. Note that the domains of these relations for fixed r but varying s are disjoint because any given access has exactly one next access, which may be an instance of any, but only one, reference s. The *forward* reuse distance, i.e., the distance to the *next* access can then be computed as

$$F_r(\mathbf{i}) = \sum_s \left(\# \bigcup_t \{\, \mathbf{i} \to A_t(\mathbf{k}) \mid \exists \mathbf{j} \in S_s, \mathbf{k} \in S_t : \mathbf{i} \to \mathbf{j} \in R_{r\to s} \wedge \mathbf{i} \preccurlyeq \mathbf{k} \preccurlyeq \mathbf{j} \,\} \right), \tag{4.5}$$

where both s and t range over all references and A_t is the access relation of reference t. The *backward* reuse distance can be defined in a similar way.

Let us now consider how one of the relations in Equation 4.5 might be constructed using primitive operations. We first construct a relation that maps the domain elements of $R_{r\to s}$ to later iterations of t, i.e.,

$$R_1 = (\operatorname{dom} R_{r\to s} \to S_t) \cap L_{r,t},$$

with $L_{r,t}$ the lexicographic order on r and t. This computation yields the relation

$$R_1 = \{\, \mathbf{i} \to \mathbf{k} \mid \mathbf{i} \in \operatorname{dom} R_{r\to s} \wedge \mathbf{k} \in S_s \wedge \mathbf{i} \preccurlyeq \mathbf{k} \,\}.$$

We can similarly construct a relation that maps the range elements to earlier iterations and then pull it back to the domain of $R_{r\to s}$. That is, we compute

$$R_2 = \left((\operatorname{ran} R_{r\to s} \to S_t) \cap L_{t,s}^{-1} \right) \circ R_{r\to s}.$$

which yields the relation

$$R_2 = \{\, \mathbf{i} \to \mathbf{k} \mid \exists \mathbf{j} \in S_s : \mathbf{i} \to \mathbf{j} \in R_{r\to s} \wedge \mathbf{k} \in S_s \wedge \mathbf{k} \preccurlyeq \mathbf{j} \,\}.$$

Finally, the relation in Equation 4.5 can be constructed as

$$A_t \circ (R_1 \cap R_2)\,.$$

4.3.4 Weighted counting

So far, we have assumed that all points in the sets we want to enumerate are equal and then the enumeration simply counts the number of elements. In some cases, however, different points may have different *weights*

and then we want to compute the sum of the weights of all elements in the set. For example, consider once more the loop nest in Figure 4.1. Let us now first compute the number of iterations of the `j`-loop for any iteration of the `i`-loop:

```
card [n] -> { [i] -> [j] : 1 <= i <= n and 1 <= j <= i };
```

The result is

```
[n] -> { [i] -> i : i <= n and i >= 1 }
```

This function assigns the weight i to each iteration i of the outer loop. The total number of iterations is then given by the sum of all these weights over all iterations of the outer loop:

```
sum [n] -> { [i] -> i : i <= n and i >= 1 };
```

and the result is

```
[n] -> { (1/2 * n + 1/2 * n^2) : n >= 1 }
```

As expected, this result is the same as that computed before in Equation 4.1. In this example, there is then also no point in performing the computation incrementally, because the results of one counting problem are used directly as the input of the next counting problem. There are some applications, however, where the result of one counting problem is first manipulated using some other operations before being used as an input to a second counting problem. For example, the input to the second counting problem may be the sum of several counting problems and/or a reformulation in terms of different variables. We will come across such an example in Section 4.4.4. As we will see below, there are also some applications where the counting problem is naturally weighted.

Since the output of an (unweighted) counting problem is a piecewise step-polynomial, the input of a weighted counting problem should be generic enough to include such piecewise step-polynomials. Similarly, since an unweighted counting problem is a special case of a weighted counting problem with weights 1, so should the output of the weighted counting problem. In fact, both input and output are exactly piecewise step-polynomials. More precisely,

Operation 4.3 (Weighted counting)
Input: a piecewise step-polynomial $q : \mathbb{Z}^n \times \mathbb{Z}^d \to \mathbb{Q}$, with (disjoint) cells K_i and associated step-polynomials q_i
Output: a piecewise step-polynomial

$$r : \mathbb{Z}^n \to \mathbb{Q} : (\mathbf{s}) \mapsto r(\mathbf{s}) = \sum_{\mathbf{t} \in \operatorname{dom} q(\mathbf{s})} q(\mathbf{s}, \mathbf{t}) = \sum_i \sum_{\mathbf{t} \in K_i(\mathbf{s})} q_i(\mathbf{s}, \mathbf{t})$$

Note that the final equality only holds if the cells K_i are indeed disjoint.

```
p = a;
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p += j * ((j-i)/4);
        *p = hard_work(i,j);
    }
```

Figure 4.7 Pointer conversion example.

Example 4.6 Consider the program in Figure 4.7. We would like to parallelize this code, but there is a (false) dependence through the pointer `p` because it is updated in every iteration and so each iteration depends on the previous iteration. The dependence can be removed by computing the sum of all updates in any previous iteration, i.e.,

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \preccurlyeq (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

with $S = \{ (i', j') \in \mathbb{Z}^2 \mid 0 \leq i' < N \wedge i' \leq j' < N \}$, which is clearly a weighted counting problem.

Example 4.7 Consider the program in Figure 4.8, where first some memory is allocated, then some other code is executed and finally the memory is freed. Suppose we want to know how much memory is allocated in total. Each iteration of the first loop allocates an amount of memory that depends on the values of the iterators. The total amount of memory can then again be computed as a weighted counting problem,

$$T(N) = \sum_{(i,j) \in S} (ij + i - N + 1),$$

with $S = \{ (i, j) \in \mathbb{Z}^2 \mid 0 \leq i < N \wedge i \leq j < N \}$.

```
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        p[i][j] = malloc(i * j + i - N + 1);
/* ... */
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        free(p[i][j]);
```

Figure 4.8 Memory allocation example.

4.4 Memory Requirement Estimates Based on Maximization Problems

4.4.1 Introduction

Recall that in this chapter, we focus on parametric estimation where some parameters values stay entirely or partially unknown while estimating the maximum required memory amount. For nonparametric cases, the reader is invited to refer to Chapter **??**, although the techniques presented in this chapter can also be applied efficiently for nonparametric cases.

Au: Please supply.

As it was presented in the previous section, many memory requirement estimation problems can be handled using the same common strategy: compute the number of elements that satisfy some conditions and then compute an upper bound of the resulting expression. For instance, the data storage requirements of some loop nests can be evaluated by first determining the amount of memory "in use" at a given point during the execution of the program and then by computing the maximum of the resulting expression over all such points. The memory in use at a given loop iteration is expressed as a piecewise step-polynomial in both the loop iterators and the structural parameters. The problem of calculating the memory requirements of a program then reduces to computing the maximum of such a polynomial over all integer points contained in parametric polytopes, resulting in an expression that only depends on the structural parameters. Notice that the general problem of maximizing a polynomial over a parametric polytope also has applications in extending static analysis beyond the polytope model [15].

As a typical example of the kind of memory requirement estimation problems that can be handled, we consider the problem of finding the maximal number of live elements during the course of a program, where an element is "live" at a given point in the program if it has been defined (written) and still needs to be used (read). A bound on the maximal number of live elements is an indication of the amount of memory required for the execution of the program.

Example 4.8 Consider the code fragment of Figure 4.9 where each statement has been labeled `Sx`, and assume that array `t` is a temporary array that is only used inside the given loop nests. The number of live elements of array `t` is obviously all the `n` elements, since every element is updated in the first loop and read in the last loop. However, we use this simple example to show how such a problem must generally be handled.

Each array element that is defined in the first loop is read and written several times in the second loop nest, and read exactly once in the last loop. As explained in Section 4.3.2, the number of live elements has to be computed for any iteration of the statements of each loop nest. The memory requirement

```
for (i = 0; i < n; ++i)
    S1: t[i] = a[i];
for (i = 0; i < n; ++i)
    for (j = 0; j < n - i; ++j)
        S2: t[j] = f(t[j], t[j+1]);
for (i = 0; i < n; ++i)
    S3: b[i] = t[i];
```

Figure 4.9 Two nested loops with temporary array `t`.

is then estimated by computing an upper bound of this numbers over the entire execution of the program.

The three iteration domains defined by each statement are:

$$S_1 = \{i \mid 0 \leq i < N\}$$
$$S_2 = \{(i,j) \mid 0 \leq i < N, 0 \leq j < n - i\}$$
$$S_3 = \{i \mid 0 \leq i < N\}$$

By using the first way of computing the number of live elements (Section 4.3.2), we first determine for each read of an array element, what is the previous write of the same element. Such a relation represents a data dependence between the update of an element and its use (a *flow dependence*, Section 4.2.2). For each read, the corresponding last previous write of the same element occurs at a previous iteration which is the last executed iteration between all the previous iterations where writes of the same element occur.

For instance, consider the reading access `t[j]` in statement `S2`. For a given value of `j`, let us call it `j'`, array element `t[j']` is accessed when `j = j'`, i.e., at every iteration (i, j') with $0 \leq i < n$. Previous writes of `t[j']` occur through the writing access `t[j]` in `S2` and `t[i]` in `S1`. In the same way as for the reading access `t[j]`, array element `t[j']` is updated at every iteration (i, j') with $0 \leq i < n$, while it is updated only once through statement `S1` when $i = j'$. Hence for any read of an element `t[j']` through reference `t[j]` in `S2` at a given iteration (i', j'), the previous writes are given by:

$$\{i \in S_1 \mid i = j'\} \cup \{(i,j) \in S_2 \mid j = j', (i,j) \prec (i',j')\}$$

It is obvious that the last previous write is performed through statement `S1` at iteration $i = j'$ if $i' = 0$ and through statement `S2` at iteration $(i = i' - 1, j = j')$ if $i' > 0$. Then we can deduce the relations linking the corresponding writes and reads in the following way:

$$R_1 = \{j \rightarrow (0,j) \mid j \in S_1 \wedge (0,j) \in S_2\}$$
$$R_2 = \{(i-1,j) \rightarrow (i,j) \mid (i,j) \in S_2 \wedge i > 0\}$$

Having determined in the same way the previous last writes (the sources) for all the remaining reads (the sinks), the next step is to count the sources and the sinks that precede a given iteration, and make the difference between both. For instance, if we only consider the reading accesses `t[j]` in statement `S2` and their sources, this computation would be, for any iteration (i', j') of the second loop nest:

$$\#\{i \in \operatorname{dom} R_1\} - \#\{(j, 0, j) \in R_1 \mid (0, j) \prec (i', j')\} \text{ if } i' = 0$$

$$\#\{(i, j) \in \operatorname{dom} R_2 \mid (i, j) \prec (i', j')\} - \#\{(i, j, k, l) \in R_2 \mid (k, l) \prec (i', j')\} \text{if } i' > 0$$

resulting in the number of live elements for any iteration (i, j) in $S_2 = n - i$. Hence the maximal number of live elements is equal to n.

The whole analysis considering all the statements and accesses to array elements can be achieved using `iscc`:

- Definition of the iteration domains:

```
D := [n] -> { S1[i] : 0 <= i < n;
              S2[i,j] : 0 <= i < n and 0 <= j < n - i;
              S3[i] : 0 <= i < n };
```

- Definitions of the maps associated to the write accesses and the maps associated to the read accesses:

```
W := { S1[i] -> t[i]; S2[i,j] -> t[j]; S3[i] -> b[i] } * D;
R := { S1[i] -> a[i]; S2[i,j] -> t[j]; S2[i,j] -> t[j+1];
       S3[i] -> t[i] } * D;
```

- Mapping of the statements onto a common iteration space:

```
S := { S1[i] -> [0,i,0]; S2[i,j] -> [1,i,j]; S3[i] -> [2,i,0] };
```

- Computing the relations between the writes and all corresponding reads:

```
Dep := (last W before R under S)[0];
```

- Computing the last reads:

```
M := (lexmax (Dep . S)) . S^-1;
```

- Setting the lexicographic order:

```
LGT := S >> S;
```

- Computing the number of live elements for any iteration (i, j):

```
Live := (card (LGT * (D -> (dom M)))) - (card ((LGT . (M^-1)) * D));
Live;
```

- Resulting in:

```
[n] -> { S2[i, j] -> n : j <= -1 + n - i and i >= 2 and j >= 1;
         S2[i, j] -> ((1/2 + n) * i - 1/2 * i^2) :
                     i = 1 and j <= -2 + n and j >= 1;
```

```
S2[i, j] -> n : j = 0 and i <= -1 + n and i >= 2;
S2[i, j] -> ((1/2 + n) * i - 1/2 * i^2) :
            i = 1 and j = 0 and n >= 2;
S2[i, j] -> n : i = 0 and j <= -1 + n and j >= 1;
S2[i, j] -> n : i = 0 and j = 0 and n >= 1;
S1[i] -> i : i <= -1 + n and i >= 1;
S3[i] -> (n - i) : i >= 2 and i <= -1 + n and n >= 2;
S3[i] -> (-1 + n) : i = 1 and n >= 2;
S3[i] -> n : i = 0 and n >= 2;
S3[i] -> 1 : n = 1 and i = 0 }
```

- Computing the maximum, *i.e.*, an upper bound of the data storage requirement:

  ```
  ub Live;
  ```

- Resulting in:

  ```
  ([n] -> { max(n) : n >= 4; max(n) : n = 3;
            max(n) : n = 2; max(n) : n = 1 }, True)
  ```

The computed bound is the exact bound since the keyword `True` is appearing in the answer, and it is obviously equal to n.

In Section 4.4.4, it is shown how dynamic memory requirements in Java programs can also be estimated using similar techniques. In the following section, an overview of the main approaches and their mathematical concepts is given. Section 4.4.3 deals with the general issue of the problem formulation.

4.4.2 Maximization of polynomials

The exact parametric maximum of polynomials over the integer points in a parametric polytope may not in general be easily computable. In the technique used in the above example [16], the problem is relaxed first by computing the maximum over all *rational* points instead of all integer points and second by computing an *upper bound* rather than the maximum. This approach consists in an extension of *Bernstein expansion* [17–19] to parametric polytopes to compute these upper bounds. The resulting upper bounds are usually fairly accurate and it can be *detected* whether the actual maximum has been computed or not.

Some other techniques to handle polynomials have been proposed by Maslov and Pugh [20], Blume and Eigenmann [21,22], and Van Engelen et al. [23]. However, these techniques are either strongly restrictive or produce less accurate estimations than the ones produced by the technique presented in [16] and based on Bernstein expansion.

4.4.2.1 Bernstein expansion

Bernstein expansion allows for the determination of bounds on the range of a multivariate polynomial considered over a box [15,24,25]. Numerical

applications of this theory have been proposed to the resolution of systems of strict polynomial inequalities [26,27]. A symbolic approach to Bernstein expansion used in program analysis has also been proposed in [15]. It has been shown that Bernstein expansion is generally more accurate than classic interval methods [28]. Moreover, in [29], Stahl has shown that for *sufficiently small* boxes, the exact range is obtained.

Bernstein polynomials are particular polynomials that form a basis for the space of polynomials. Hence any polynomial can be expressed in this basis through coefficients, the Bernstein coefficients, that have interesting properties and that can be computed through a direct formula. Due to the Bernstein convex hull property [30], the value of the polynomial is then bounded by the values of the minimum and maximum Bernstein coefficients. The direct formula allows symbolic computation of these Bernstein coefficients giving a supplementary interest to the use of this theory [15,16].

4.4.2.2 Symbolic range propagation

Symbolic range propagation [21,22] is certainly the most commonly used approach due mainly to its relative simplicity. Even if it usually provides less accurate results than with Bernstein expansion, it can still be useful while considering complex problems inducing many iteration domains and parameters. Indeed, the computation time in such cases can be huge with the Bernstein approach, since it needs several complex computations on parametric polytopes.

In this technique, it is assumed that each variable has a (symbolic) lower and upper bound and these ranges are repeatedly substituted in the polynomial. In each iteration, the expression is simplified using a set of rewrite rules. If a variable occurs multiple times in the same expression, then overly conservative bounds can be generated. However, if it can be determined, by recursively applying the algorithm to the first order forward difference of the polynomial, that the expression is monotonically nonincreasing or nondecreasing in a given variable, then the lower and upper bounds of the variable can safely be substituted simultaneously in the whole expression, leading to a tighter bound. The main disadvantage of this technique is that the accuracy can be very low for non-monotonic polynomials.

4.4.3 General problem formulation

When handling a memory requirement estimation problem, the essential task is to translate it conveniently into a counting problem followed by a maximization problem.

We are interested in the case where the elements of counting are integer vectors and the conditions can be described by *linear* constraints. The first step is then to compute the number of elements $f(\mathbf{p}, \mathbf{q})$ of some set $S(\mathbf{x}, \mathbf{p}, \mathbf{q})$, i.e.,

$$f(\mathbf{p}, \mathbf{q}) = \#\{\mathbf{x} \in \mathbb{Z}^n \mid \exists \mathbf{y} \in \mathbb{Z}^{n'} : p(\mathbf{x}, \mathbf{y}, \mathbf{p}, \mathbf{q})\}, \tag{4.6}$$

where $p(\mathbf{x}, \mathbf{y}, \mathbf{p}, \mathbf{q})$ is a conjunction of m linear constraints on $\mathbf{x}$, $\mathbf{y}$, $\mathbf{p}$ and $\mathbf{q}$,

$$p(\mathbf{x}, \mathbf{y}, \mathbf{p}, \mathbf{q}) \iff A\mathbf{x} + B\mathbf{y} \geq C\mathbf{p} + D\mathbf{q} + \mathbf{f},$$

with $A \in \mathbb{Z}^{m \times n}$, $B \in \mathbb{Z}^{m \times n'}$, $C \in \mathbb{Z}^{m \times r}$, $D \in \mathbb{Z}^{m \times r'}$ and $\mathbf{f} \in \mathbb{Z}^m$. In the second step, we compute an upper bound on $f(\mathbf{p}, \mathbf{q})$. That is, we compute an $U(\mathbf{q})$ such that

$$U(\mathbf{q}) \geq M(\mathbf{q}) = \max_{\mathbf{p} \in Q(\mathbf{q})} f(\mathbf{p}, \mathbf{q}) \qquad \text{for all } \mathbf{q}, \tag{4.7}$$

with Q the domain of f and where we use the shorthand $Q(\mathbf{q}) = \{\, \mathbf{p} \mid (\mathbf{p}, \mathbf{q}) \in Q \,\}$.

The first Problem (Equation 4.6) is a counting problem (see Section 4.3) while the second problem (Equation 4.7) is a "maximization" problem. Both of these problems are *parametric*, i.e., the result is not simply a number, but rather an expression in a number of parameters. The variables that act as parameters in one problem are, however, not the same as those that act as parameters in the other problem. In general, we can identify fours sets of variables in the two problems:

- The variables that are existentially quantified in the counting problem Equation 4.6; in the example of Section 4.4.1, there are no such variables.
- The elements that need to be counted; in the example, these are the indices of the array or the iteration in which they are defined or used.
- The variables over which the maximum needs to be taken; in the example these are the iterations of $D_2(n)$.
- The structural parameters; in the example, there is a single structural parameter `n`.

The latter two sets of variables will be parameters for the counting problem, while only the structural parameters will be parameters for the maximization problem.

If there are no existentially quantified variables $\mathbf{y}$, i.e., if $n' = 0$, then the set in Equation 4.6 is a *parametric polyhedron* (Section 4.2). If the polyhedron is bounded for each value of the parameters (which will usually be the case when we want to count the number of integer points in the polyhedron), then the set is a *parametric polytope* (Section 4.2).

As it was presented in Section 4.3, the number of integer points in such parametric polytopes are piecewise step-polynomials:

$$f(\mathbf{p}, \mathbf{q}) = \begin{cases} f_1(\mathbf{p}, \mathbf{q}) & \text{if } (\mathbf{p}, \mathbf{q}) \in Q_1 \\ \dots & \\ f_M(\mathbf{p}, \mathbf{q}) & \text{if } (\mathbf{p}, \mathbf{q}) \in Q_M, \end{cases} \tag{4.8}$$

i.e., a subdivision of the parameter space Q (of the counting problem), with a step-polynomial $f_i(\mathbf{p}, \mathbf{q})$ associated to each cell Q_i of the subdivision. These

piecewise step-polynomials that result from counting problems are also called Ehrhart polynomial by some authors, e.g., [31]. Notice that the cells Q_i in the subdivision are themselves polyhedra.

To compute $U(\mathbf{q})$ in Equation 4.7, we first compute

$$U_i(\mathbf{q}) \geq M_i(\mathbf{q}) = \max_{\mathbf{p} \in Q_i(\mathbf{q})} f_i(\mathbf{p}, \mathbf{q}) \qquad \text{for all } \mathbf{q}.$$

Note that Q_i is interpreted here as a parametric polyhedron with only the $\mathbf{q}$ as parameters. As in the counting problem, we may assume that Q_i is a parametric *polytope*. Finally $U(\mathbf{q})$ is constructed such that

$$U(\mathbf{q}) \geq U_i(\mathbf{q}) \qquad \text{for all } i \text{ and for all } \mathbf{q}. \tag{4.9}$$

4.4.4 Estimating dynamic memory requirements

The techniques mentioned so far can be also used to compute parametric bounds of dynamic memory requirements. As an example we will analyze a technique that computes dynamic memory requirements for Java programs [32].

Given a method `m` with parameters $p_1, \ldots, p_k$, the technique computes a parametric polynomial in $p_1, \ldots, p_k$ over-approximating the amount of dynamic memory *required* to execute `m`.

Java, is an object oriented language with automatic memory management. Memory allocation is controlled by the programmer which creates objects when executing a `new` statement. Memory deallocation is not controlled by the programmer but by a special agent, a *garbage collector* (GC), which take cares of collecting objects when they are not longer referenced.

In order to compute accurate bounds it is very important to analyze program allocations and deallocations. Therefore, an analysis should consider both the program and GC behaviors. One alternative used to approximate GC behavior is to statically compute object lifetimes (a sequence of program statements starting at the moment of object creation and finishing at the point when the object can be collected) in order to predict at compile time, for every object, where it can be collected. This can be performed by the aid of escape analysis techniques [33,34] or by performing region synthesis [35,36] which associate the lifetime set of objects to the lifetime of computation units (e.g. methods, classes, threads, etc).

Here we will adopt the notion regions by associating object lifetimes to methods. We will use $\text{ret}_{\texttt{m}}$ to denote the size of the objects returned (or escaping) a method `m`. That is, the objects that should live even when method `m` finishes its execution. We will use $\text{cap}_{\texttt{m}}$ to denote the size of the objects allocated by method `m` that can be safely collected at the end its method `m` execution. Another way to see $\text{cap}_{\texttt{m}}$ is thinking they are auxiliary objects created during method `m` execution which are not longer required by a caller of `m`. The amount

```
void m0(int m) {
  for(c=1;c<= m;c++) {
    m1(c);
    B[] m2Arr=m2(2*m-c);
  }
}
B[] m2(int n) {
  B[] arrB = new B[n];
  for(j=1;j<= n;j++) {
    B b=new B();
    C c=new c();
    arrB[j-1]=b;
  }
  return arrB;
}
```

```
void m1(int k) {
  for(i=1;i<=k;i++) {
    A a = new A();
    B[] captArr=m2(i);
  }
}
```

Figure 4.10 Dynamic memory allocation example.

of memory required to run a method m (denoted $\text{memRq}_{\texttt{m}}$) is composed by both $\text{cap}_{\texttt{m}}$ and $\text{ret}_{\texttt{m}}$. since in order to run m the system will require enough memory to allocate the objects that will be created by m and collected when it finishes ($\text{cap}_{\texttt{m}}$) and the objects allocated by m which live longer ($\text{ret}_{\texttt{m}}$). Nevertheless, we will see later that objects returned by methods tend to be captured by other methods in the call stack, meaning that eventually returned objects are considered when computing captured objects. Therefore, for convenience, we will consider only captured objects when computing $\text{memRq}_{\texttt{m}}$ and we will add $\text{ret}_{\texttt{m}}$ to the estimation only when m represents the application's *main* method.

Consider the example of Figure 4.10 and assume for simplicity that all objects are of size 1. Method `m2` does not call any other methods. All allocations assigned to variable `c` can be captured by `m2` since they are neither returned nor linked to parameters or static variables. The method returns `arrB` which made the array and objects assigned to variable `b` live longer than method `m2`. By describing the iteration space of the loop by $1 \leq j \leq n$ we can apply a parametric counting technique to count the number of visits to each `new` statement and approximate its consumption. We therefore have:

$$\text{cap}_{\texttt{m2}}(n) = \#\{\,(j) \mid 1 \leq j \leq n\} = n$$

$$\text{ret}_{\texttt{m2}}(n) = n + \#\{\,(j) \mid 1 \leq j \leq n\} = 2n$$

$$\text{memRq}_{\texttt{m2}}(n) = \text{cap}_{\texttt{m2}}(n) = n.$$

Note that we assume here that n is nonnegative.

Method `m1` does call another method, namely `m2`, and it captures all the memory it allocates itself as well as the memory that escaped (and was returned) from `m2`. Note that `m2` is called several times within a loop which iteration space which can be modeled by $(1 \leq i \leq k)$. Escaped objects are accumulative meaning the space for the objects allocated and returned by `m2` at *each* iteration has to be considered when computing `m1` requirements.

To compute the amount of memory required to run `m1` we need to consider its allocations (assigned to variable `a`) and the allocations performed by the calls `m2`. Method `m2`, in addition to returned objects, requires space for the i objects it captures. Since they are released at the end of its execution, it is only needed to consider the space for the call to `m2` that consumes most auxiliary objects.

Therefore, we have:

$$\begin{aligned} \mathrm{ret}_{\mathtt{m1}}(k) &= 0 \\ \mathrm{cap}_{\mathtt{m1}}(k) &= \#\{\,(i) \mid 1 \leq i \leq k\} + \sum_{1 \leq i \leq k} (\mathrm{ret}_{\mathtt{m2}}(i)) \\ &= k + \sum_{1 \leq i \leq k} (2i) = k + k^2 + k = k^2 + 2k \end{aligned}$$

$$\begin{aligned} \mathrm{memRq}_{\mathtt{m1}}(k) &= \mathrm{cap}_{\mathtt{m1}}(k) + \max_{1 \leq i \leq k} \mathrm{memRq}_{\mathtt{m2}}(i) \\ &= k^2 + 2k + k = k^2 + 3k \end{aligned}$$

where Bernstein expansion is used to compute $\max_{1 \leq i \leq k} \mathrm{memRq}_{\mathtt{m2}}(i)$ and weighted counting is used to compute the sum. We again assume that $k \geq 0$.

Finally, notice that method `m0` calls methods `m1` and `m2` within a loop. We have to consider the objects escaping from both `m1` and `m2`:

$$\mathrm{ret}_{\mathtt{m0}}(m) = 0$$

$$\mathrm{cap}_{\mathtt{m0}}(m) = \sum_{1 \leq c \leq m} \mathrm{ret}_{\mathtt{m1}}(c) + \mathrm{ret}_{\mathtt{m2}}(2m - c) = \sum_{1 \leq c \leq m} 0 + 2(2m - c) = 3m^2 - m$$

Similarly, to compute the actual requirements for `m0` we need to observe that the space reserved for objects from method `m1` and `m2` can be shared. Method `m0` first calls to method `m1` and, when it returns, then it calls `m2`. Objects required for `m1` are not longer needed when it finished its execution,

releasing space for objects allocated when executing `m2`. Thus, it is enough to consider only the maximum between requirements for `m1` and `m2`:

$$
\begin{aligned}
&\text{memRq}_{\texttt{m0}}(\text{m}) \\
&\quad = \text{cap}_{\texttt{m0}}(m) + \max\left(\max_{1\le c\le m} \text{memRq}_{\texttt{m1}}(c), \max_{1\le c\le m} \text{memRq}_{\texttt{m2}}(2m-c)\right) \\
&\quad = \text{cap}_{\texttt{m0}}(m) + \max\left(\max_{1\le c\le m} c^2+3c, \max_{1\le c\le m} 2m-c\right) \\
&\quad = 3m^2 - m + \max\left(m^2+3m, 2m-1\right) \\
&\quad = 3m^2 - m + m^2 + 3m \\
&\quad = 4m^2 + 2m,
\end{aligned}
$$

The general solution is

$$
\text{memRq}_{\texttt{m0}}(m) = \begin{cases} 4m^2+2m & \text{if } m \ge 1 \\ 0 & \text{if } m \le 0. \end{cases}
$$

In general, using this notion of captured and returned (escaping) objects, the memory requirements of a method can be computed in terms of memory requirements of the methods it calls. Therefore:

$$
\text{memRq}_{\texttt{m}}(\bar{p}) = \text{cap}_{\texttt{m}}(p_m) + \max_{m' \text{called by } m}\left(\max_{inv^m_{m'}} \text{memRq}_{\texttt{m'}}(\bar{a}^m_{m'})\right)
$$

where $\bar{p}$ are methods `m` formal parameters, $inv^m_{m'}$ is the iteration space defined in the call from method `m` to method `m'` and $\bar{a}^m_{m'}$ are the arguments used in that call.

This analysis can be automated using `iscc`. Notice there are many ways to solve this problem using the calculator (e.g., calling it several times for each symbolic operation, generating a set of operations per method). Here we propose one that make extensive use of `iscc` symbolic capabilities and generate all the results in one script.

- Analysis of method `m2`. Definition of iteration space for the loop:
  `D_L_m2 := {[n] -> [j] : 1<=j<=n};`
  The number of objects assigned respectively to `c` and `b` are computed counting the number of integer solutions.
  `alloc_b := card D_L_m2; alloc_c := card D_L_m2;`
  which is `{[n] -> n : n>=1};`

For the array we use its dimension:
`alloc_bArr := {[n] -> n : n>=1};`
Thus, the amount of objects captured, returned and required are:
`cap_m2 := alloc_c;`
which is `{[n] -> n : n >= 1 }`
`ret_m2 := alloc_b + alloc_bArr;`
which is `{[n] -> 2 * n : n >= 1 }`
`memReq_m2:= cap_m2 + {[n] -> max(0) };`
which is `{[n] -> max(n) : n >= 1}` (we convert `memReq_m2` to the result of a maximization operation for the sake of compositionally)

- Analysis of method `m1`. The iteration space for the loop is:
  `D_L_m1 := {[k] -> [i] : 1<=i<=k};`
  Then, the numbers of objects assigned to `a` are:
  `alloc_a := card D_L_m1;`
  This relation represents the call `m2(i)` in the loop:
  `call_m1m2 := (domain_map D_L_m1)^-1 . { [[k]->[i]] -> [i] };`
  The objects returned from `m2` are accumulated:
  `capRet_m2:= call_m1m2 . ret_m2;`
  `cap_m1 := alloc_a + capRet_m2;`
  which is `{[k] -> (2 * k + k^2) : k >= 1 }`
  `ret_m1 := {[k] -> 0};`
  And the object required by `m2` but collected by itself are maximized (notice that `memReq` is a fold):
  `max_m1m2 := call_m1m2 . memReq_m2;`
  `memReq_m1 := cap_m1 + max_m1m2;`
  which is `{[k] -> max((3 * k + k^2)) : k >= 1 }`
- Analysis of method `m0`. The iteration space for the loop is:
  `D_L_m0 := {[m] -> [c] : 1 <= c <= m };`
  These are the relations for the calls `m1(c)` and `m2(2m-c)` in the loop.
  `call_m0m1 := (domain_map D_L_m0)^-1 . { [[m]->[c]] -> [c] };`
  `call_m0m2 := (domain_map D_L_m0)^-1 . { [[m]->[c]] -> [2m-c]};`
  Then:

  ```
  capRet_m1:= call_m0m1 . ret_m1;
  capRet_m2:= call_m0m2 . ret_m2;
  max_m0m1 := call_m0m1 . memReq_m1;
  max_m0m2 := call_m0m2 . memReq_m2;
  ```

  Thus, the amount of objects captured, returned and required are:
  `retM0 := {[m] -> 0};`
  `cap_m0 := capRet_m1 + capRet_m2;`
  which is `{[m] -> (-m + 3 * m^2) : m >= 1}`
  `memReq_m0 := cap_m0 + (max_m0m1 . max_m0m2);`
  which is `{[m] -> max((2 * m + 4 * m^2)) : m >= 1}` as we manually computed.

4.4.5 Software implementation

The computation of lower and upper bounds on an arbitrary multivariate piecewise step-polynomials, defined over linearly parametrized convex polytopes, has been implemented in the *Integer Set Library (isl)* (Section 4.2.3.5). This includes the computation of the parametric Bernstein coefficients of the polynomials and a procedure reducing the number of potential results. These latter computations were initially implemented as a standalone application named *bernstein* [16].

4.5 Conclusion

Memory requirement evaluation of applications is a major issue in the design of computer systems, and specifically in the case of embedded systems. Moreover, evaluation results, when parametrized, can cover all possible execution configurations from only one unique program analysis process. We have shown for several application examples that this problem can often consist in maximizing a parametric and multivariate polynomial defined over a parametric convex domain. We proposed the use of some advanced mathematical tools to compute accurate bounds for such polynomials, and even exact bounds in some cases. All these tools have been implemented and are freely available.

References

1. David Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, August 1995.
2. Rachid Seghir and Vincent Loechner. Memory optimization by counting points in integer transformations of parametric polytopes. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea*, October 2006.
3. Sven Verdoolaege, Kevin M. Woods, Maurice Bruynooghe, and Ronald Cools. Computation and manipulation of enumerators of integer projections of parametric polytopes. Report CW 392, Dept. of Computer Science, K.U.Leuven, Leuven, Belgium, 2005.
4. Bernard Boigelot and Louis Latour. Counting the solutions of Presburger equations without enumerating them. *Theoretical Computer Science*, 313(1):17–29, February 2004.

5. Erin Parker and Siddhartha Chatterjee. An automata-theoretic algorithm for counting solutions to Presburger formulas. In *Compiler Construction 2004*, volume 2985 of *Lecture Notes in Computer Science*, pages 104–119, Berlin, April 2004. Springer-Verlag.
6. Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *The Journal of Symbolic Computation*, 38(4):1273–1302, 2004.
7. William Pugh. Counting solutions to Presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 121–134, 1994.
8. Philippe Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, 1996.
9. A. Barvinok and J. Pommersheim. An algorithmic theory of lattice points in polyhedra. *New Perspectives in Algebraic Combinatorics*, 38:91–147, 1999.
10. S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.
11. Sven Verdoolaege, Kristof Beyls, Maurice Bruynooghe, and Francky Catthoor. Experiences with enumeration of integer projections of parametric polytopes. In Rastislav Bodík, editor, *Proceedings of 14th International Conference on Compiler Construction, Edinburgh, Scotland*, volume 3443 of *Lecture Notes in Computer Science*, pages 91–105, Berlin/Heidelberg, 2005. Springer.
12. Matthias Köppe, Sven Verdoolaege, and Kevin M. Woods. An implementation of the barvinok–woods integer projection algorithm. In Matthias Beck and Thomas Stoll, editors, *The 2008 International Conference on Information Theory and Statistical Learning*, July 2008.
13. Sven Verdoolaege and Maurice Bruynooghe. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In Matthias Beck and Thomas Stoll, editors, *The 2008 International Conference on Information Theory and Statistical Learning*, July 2008.
14. Kristof Beyls and Erik D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.
15. Ph. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In Evelyn Duesterwald,

editor, *13th International Conference on Compiler Construction, CC 2004*, volume 2985 of *LNCS*, pages 120–133. Springer, April 2004.

16. Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17:983–996, 08 2009.

Au: Should it be 2008, 2009?

17. S. Bernstein. *Collected Works*, volume 1. USSR Academy of Sciences, 1952.
18. S. Bernstein. *Collected Works*, volume 2. USSR Academy of Sciences, 1954.
19. M. Zettler and J. Garloff. Robustness analysis of polynomials with polynomial parameter dependency using bernstein expansion. *IEEE Transactions on Automatic Control*, 43(3):425–431, 1998.
20. Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. In *CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing*, September 1994.
21. William Blume and Rudolf Eigenmann. Symbolic range propagation. Technical Report 1381, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, October 1994.
22. William Blume and Rudolf Eigenmann. Symbolic range propagation. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 357–363, Washington, DC, USA, 1995. IEEE Computer Society.
23. R. A. Van Engelen, K. Gallivan, and B. Walsh. Parametric timing estimation with the Newton-Gregory formulae. *Journal of Concurrency and Computation: Practice and Experience*, 18(10):1434–1464, September 2006.
24. Jakob Berchtold and Adrian Bowyer. Robust arithmetic for multivariate bernstein-form polynomials. *Computer-aided Design*, 32:681–689, 2000.
25. R.T. Farouki and V.T. Rajan. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design*, 4(3):191–216, 1987.
26. J. Garloff. Application of bernstein expansion to the solution of control problems. In J. Vehi and M. A. Sainz, editors, *Proceedings of MISC'99 - Workshop on Applications of Interval Analysis to Systems and Control*, pages 421–430. University of Girona, Girona (Spain), Springer Netherlands, 1999.

27. J. Garloff and B. Graf. *The Use of Symbolic Methods in Control System Analysis and Design*, chapter Solving Strict Polynomial Inequalities by Bernstein Expansion, pages 339–352. Institution of Electrical Engineers (IEE), London, 1999.
28. R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design*, 19:553–587, 2002.
29. V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. PhD thesis, Johannes Kepler University Linz, Austria, 1995.
30. G. Farin. *Curves and Surfaces in Computer Aided Geometric Design*. Academic Press, San Diego, 1993.
31. Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, volume 19(2), Kluwer Academic, 1998.
32. Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements, June 2008.
33. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23. ACM Press, 2001.
34. B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.
35. Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *ISMM '04: Proceedings of the 4th International Symposium on Memory Management*, pages 85–96, New York, NY, USA, 2004. ACM Press.
36. Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in java. In *RV 2004: International Workshop on Runtime Verification*, volume 113 of *ENTCS*, pages 105–121, Barcelona, Spain, April 2004. ETAPS, Elsevier.

Au: Please supply book/journal title.