Symbolic Polynomial Maximization over Convex Sets and its Application to Memory Requirement Estimation

PHILIPPE CLAUSS
Université Louis Pasteur, France
and
FEDERICO JAVIER FERNÁNDEZ and DIEGO GARBERVETSKY
Universidad De Buenos Aires, Argentina
and
SVEN VERDOOLAEGE
Universiteit Leiden, The Netherlands

Memory requirement estimation is an important issue in the development of embedded systems, since memory directly influences performance, cost and power consumption. So it is 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. In this paper, we propose an original approach based on the theory of Bernstein expansion allowing the resolution of many important memory issues that are expressed as the problem of maximizing a parametric polynomial defined over a parametric convex domain. The paper is illustrated with several application examples.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Memory management, Optimization*

General Terms: Design, Languages, Measurement, Performance, Verification

Additional Key Words and Phrases: Bernstein expansion, convex polytopes, memory requirement, static program analysis, program optimization

1. INTRODUCTION

The determination of the amount of memory required by a program through static analysis has received a lot of attention in recent years [Verbauwhede et al. 1994; Grun et al. 1998; Zhao and Malik 2000; Ramanujam et al. 2001; Kjeldsberg et al.

Author's address: Ph. Clauss, ICPS-LSIIT, Université Louis Pasteur, Pôle API, Boul. S. Brant, 67400 Illkirch, France.
F. J. Fernández and D. Garbervetsky, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad De Buenos Aires, Argentina
S. Verdoolaege, Leiden Institute of Advanced Computer Science, Universiteit Leiden, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

2004; Zhu et al. 2006]. Usually, the first step is to determine the amount of memory "in use" at a given point during the execution of the program. The memory requirement is then obtained by computing the maximum of the resulting expression over all such points.

In particular, if the program consists of a sequence of loop nests with loop bounds and array references that are affine functions of the enclosing loop iterators and structural parameters, then the iterations of the loops can be represented by the integer points in parametric polytopes. This representation is known as the polytope model [Feautrier 1996]. The memory in use at a given loop iteration is or can be approximated by a 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 a polynomial over all integer points in a parametric polytope, resulting in an expression that only depends on the structural parameters. This maximization of a polynomial over a parametric polytope also has applications in extending static analysis beyond the polytope model [Clauss and Tchoupaeva 2004].

De Loera et al. [2006] have recently shown that maximizing an arbitrary polynomial over the integer points in a non-parametric polytope is NP-hard, but have also given a fully polynomial-time approximation scheme for computing this maximum when the polynomial is non-negative and the dimension of the polytope is fixed. However, to the best of our knowledge, their algorithm has not been implemented yet and it cannot easily be extended to the parametric case. This evidence suggests that the exact parametric maximum over the integer points in a parametric polytope may not in general be easily computable. We therefore relax our problem 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. In particular, we will use an extension of *Bernstein expansion* to parametric polytopes to compute these upper bounds. The resulting upper bounds will usually be fairly accurate and we can *detect* whether we have computed the actual maximum or not.

Bernstein expansion [Bernstein 1952; 1954] allows for the determination of bounds on the range of a multivariate polynomial considered over a box [Berchtold and Bowyer 2000; Farouki and Rajan 1987; Clauss and Tchoupaeva 2004]. Numerical applications of this theory have been proposed to the resolution of systems of strict polynomial inequalities [Garloff 1999; Garloff and Graf 1999]. A symbolic approach to Bernstein expansion used in program analysis has also been proposed by Clauss and Tchoupaeva [2004]. It has been shown that Bernstein expansion is generally more accurate than classic interval methods [Martin et al. 2002]. Moreover, Stahl [1995] 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 [Farin 1993], 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 [Clauss and Tchoupaeva 2004].

Bernstein expansion has already been used by Clauss and Tchoupaeva [2004] to handle parameterized polynomials defined over parameterized boxes. Several applications such as non-linear dependence analysis or dead code elimination are shown. However, the proposed approach is limited to domains defined as boxes. These boxes also need to be linearly transformed to unit boxes. This transformation, when applied to parameterized boxes, has to exclude some parameter values for which the transformation would yield divisions by zero. Hence the considered polynomials have to be evaluated for these specific values.

In this paper, we propose an extension of the theory of Bernstein expansion to handle parameterized multivariate polynomial expressions where the possible values of the variables are defined over parametric convex polytopes. These parametric polytopes can be described either as the convex hull of a finite set of parametric generators or as the solution set of a finite number of linear constraints over the variables and the parameters. Then we use this extension to compute upper bounds for multivariate polynomials modeling the memory usage of programs. More precisely, it is shown how the described technique can be used to compute bounds on the memory consumption of programs.

This paper is organized as follows. In Section 2, the necessary theoretical concepts used in the rest of the paper are presented. We first recall how Bernstein expansion is classically done for a polynomial defined over an interval, and then how it can be extended to polynomials defined over convex polytopes by use of the barycentric coordinates of the studied values. The general use of Bernstein expansion in some classic static analysis issues is detailed in Section 3. It is shown how accurate results are obtained for the bounds of a multivariate polynomial defined over a parametric domain and also for the bounds of the number of live elements occurring during the execution of a program. We briefly give some additional information about our software implementation in Section 4. Section 5 is devoted to the description of several interesting applications of the Bernstein approach to program analysis issues and more specifically to important memory behavior issues: the computation of the parametric memory size used by a program, of the FIFO sizes in process networks, of bounds on the data reuse distances to select efficient cache hints for load instructions and finally the estimation of dynamic memory requirements for imperative object-oriented programs. This section is illustrated with several examples. Comparisons with other related works are given in Section 6: works focusing on polynomials in program analysis and works focusing on memory requirement estimation. Finally, conclusions and further perspectives are given in Section 7.

2. SYMBOLIC BERNSTEIN EXPANSION OVER A CONVEX POLYTOPE

This section explains the theory behind Bernstein expansion. We first recall the classical Bernstein expansion of a univariate polynomial over an interval and then show how it can be extended to multivariate parametric polynomials over parametric convex polytopes.

2.1 Bernstein Expansion over an Interval

There are many ways to represent a (rational) univariate degree-d polynomial $p(x) \in \mathbb{Q}[x]$. The canonical representation of $p(x)$ is as a $\mathbb{Q}$-linear combination of the power

base, i.e., the powers of x,

$$p(x) = \sum_{i=0}^{d} a_i x^i, \tag{1}$$

with $a_i \in \mathbb{Q}$. The polynomial $p(x)$ can also be represented as a $\mathbb{Q}$-linear combination of the degree-d *Bernstein base polynomials* [Bernstein 1952; 1954; Farouki and Rajan 1987; Berchtold and Bowyer 2000]:

$$p(x) = \sum_{k=0}^{d} b_k^d B_k^d(x), \tag{2}$$

where the Bernstein polynomials $B_i^d(x)$ are defined by:

$$B_k^d(x) = \binom{d}{k} x^k (1-x)^{d-k} \quad k = 0, 1, ..., d \quad \binom{d}{k} = \frac{d!}{k!(d-k)!}, \tag{3}$$

and $b_i^d \in \mathbb{Q}$ are the Bernstein coefficients corresponding to the degree-d basis.

Example 2.1. Here is an example of a univariate polynomial in its power form and in its Bernstein form:

$$p(x) = x^3 - 5x^2 + 2x + 4 = 4B_0^3(x) + \frac{14}{3}B_1^3(x) + \frac{11}{3}B_2^3(x) + 2B_3^3(x)$$

where $B_0^3(x) = (1-x)^3$, $B_1^3(x) = 3x(1-x)^2$, $B_2^3(x) = 3x^2(1-x)$ and $B_3^3(x) = x^3$. We will explain below how to compute the Bernstein coefficients in this expression. □

The Bernstein expansion of a polynomial has many interesting properties. The properties that will interest us most here is that the sum of the Bernstein base polynomials (3) is 1 and that, on the interval $[0, 1]$, $0 \leq B_k^d(x) \leq 1$. The first property follows from the identity:

$$1 = (x + (1-x))^d = \sum_{k=0}^{d} B_k^d(x).$$

On the interval $[0, 1]$, Equation (2) expresses the polynomial $p(x)$ as a convex combination (with coefficients $B_i^d(x)$) of the Bernstein coefficients b_i^d. On this interval, the polynomial $p(x)$ is therefore bounded by its Bernstein coefficients, i.e.,

$$\min_{0 \leq i \leq d} b_i^d \leq p(x) \leq \max_{0 \leq i \leq d} b_i^d.$$

Moreover, if the minimum or maximum of the b_i^d is b_0^d or b_d^d then this bound is exact, since they correspond to values taken by $p(x)$ at the vertices as is clear from (3). These coefficients where the bound is exact are sometimes referred to as *sharp coefficients*.

Example 2.2. Figure 1 shows the polynomial $p(x) = x^3 - 5x^2 + 2x + 4$ from the previous example, the terms $b_i^3 B_i^3(x)$ of its Bernstein form and the constants b_i^3. On the interval $[0, 1]$, the polynomial is bounded by the minimal and maximal Bernstein coefficients, $b_3^3 = 2$ and $b_1^3 = 14/3$. The first of these coefficients is sharp; the second is not.

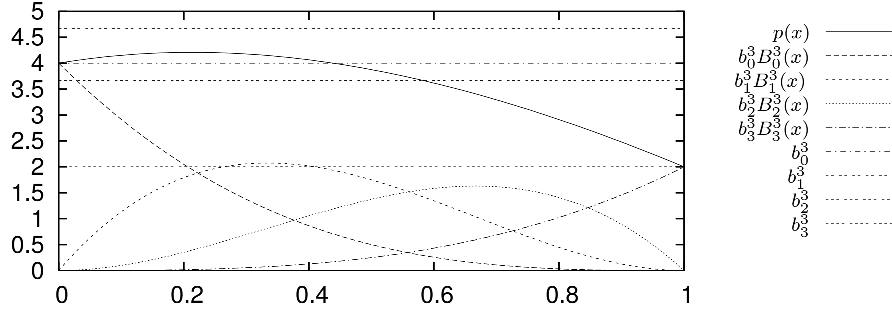

Fig. 1. Decomposition of the polynomial $p(x) = x^3 - 5x^2 + 2x + 4$ in the Bernstein basis

To compute the Bernstein coefficients b_i^d from the power form coefficients a_i, we write the point x on the interval $[0, 1]$ in terms of its barycentric coordinates,

$$x = \alpha_0\, v_0 + \alpha_1\, v_1,$$

with

$$\alpha_i \geq 0 \quad \text{for } i \in \{0, 1\} \qquad \text{and} \qquad \alpha_0 + \alpha_1 = 1$$

and where $v_0 = 0$ and $v_1 = 1$ are the vertices of the interval $[0, 1]$. We see that $\alpha_1 = x$ and $\alpha_0 = 1 - x$ and that the Bernstein base polynomials (3) are homogeneous polynomials of degree d in α_0 and α_1. To write $p(x)$ (1) as a homogeneous polynomial in α_0 and α_1, we simply substitute $x = \alpha_0\, 0 + \alpha_1\, 1 = \alpha_1$ and multiply each degree-i homogeneous component of $p(\alpha_0, \alpha_1)$ ($i \leq d$) by $1 = (\alpha_0 + \alpha_1)^{d-i}$, i.e.:

$$\begin{aligned} p(\alpha_0, \alpha_1) &= \sum_{i=0}^{d} a_i \alpha_1^i (\alpha_0 + \alpha_1)^{d-i} \\ &= \sum_{i=0}^{d} a_i \alpha_1^i \left(\sum_{j=0}^{d-i} \binom{d-i}{j} \alpha_0^{d-i-j} \alpha_1^j \right) = \sum_{k=0}^{d} \left(\sum_{i=0}^{k} a_i \binom{d-i}{k-i} \right) \alpha_1^k \alpha_0^{d-k}. \end{aligned}$$

Comparing with (2) and noting that

$$B_k^d(x) = B_k^d(\alpha_0, \alpha_1) = \binom{d}{k} \alpha_1^k (\alpha_0)^{d-k}, \tag{4}$$

we obtain:

$$b_k^d = \sum_{i=0}^{k} \frac{\binom{d-i}{k-i}}{\binom{d}{k}} a_i = \sum_{i=0}^{k} \frac{\binom{k}{i}}{\binom{d}{i}} a_i,$$

where the last equality follows from the identity:

$$\binom{d-i}{k-i} \binom{d}{i} = \binom{d}{k} \binom{k}{i}.$$

Bounds on the values attained by a polynomial over an arbitrary interval $[a, b]$ can be obtained using essentially the same technique. We write:

$$x = \alpha_0\, a + \alpha_1\, b,$$

with

$$\alpha_i \geq 0 \quad \text{for } i \in \{0, 1\} \qquad \text{and} \qquad \alpha_0 + \alpha_1 = 1,$$

substitute this expression in $p(x)$ to obtain a polynomial $p(\alpha_0, \alpha_1) \in \mathbb{Q}[\alpha_0, \alpha_1]$, multiply each term with the appropriate power of $1 = \alpha_0 + \alpha_1$ and compute the coefficients b_k^d with respect to the basis formed by the terms in the expansion

$$1 = (\alpha_0 + \alpha_1)^d = \sum_{k=0}^{d} B_k^d(\alpha_0, \alpha_1).$$

The terms $B_k^d(\alpha_0, \alpha_1)$ are defined as in (4). They are then again the coefficients in the expression of $p(\alpha_0, \alpha_1)$ as a convex combinations of the b_k^d and so

$$\min_{0 \leq i \leq d} b_i^d \leq p(x) \leq \max_{0 \leq i \leq d} b_i^d$$

on the interval $[a, b]$.

2.2 Bernstein Expansion over a Convex Polytope

In this subsection, we generalize the so-called Bernstein-Bezier form of a polynomial defined over a triangle [Farin 1993], and apply the same principles to multivariate parameterized polynomials defined over parameterized polytopes of any dimension.

A (rational) convex polytope $P \subset \mathbb{Q}^n$ is the convex hull of a set of points $\mathbf{v}_i$,

$$P = \left\{ \mathbf{x} \mid \exists \alpha_i \in \mathbb{Q} : \mathbf{x} = \sum_i \alpha_i \mathbf{v}_i, \alpha_i \geq 0, \sum_i \alpha_i = 1 \right\}.$$

If no $\mathbf{v}_i$ is a convex combination of the other $\mathbf{v}_i$ and then these $\mathbf{v}_i$ are called the vertices of the polytope.

To compute lower and upper bounds on a (rational) multivariate polynomial $p(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}] = \mathbb{Q}[x_1, \ldots, x_n]$,

$$p(x_1, x_2, \ldots x_n) = \sum_{i_1=0}^{d_1} \sum_{i_2=0}^{d_2} \cdots \sum_{i_n=0}^{d_n} a_{i_1, i_2, \ldots, i_n}\, x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \tag{5}$$

over a polytope $P \subset \mathbb{Q}^n$, we essentially follow the procedure from the previous section. We first write $\mathbf{x}$ as a convex combinations of the vertices

$$\mathbf{x} = \sum_i \alpha_i \mathbf{v}_i$$

and substitute this expression in the polynomial $p(\mathbf{x})$. We then multiply each term in the result with the appropriate power of $1 = \sum_i \alpha_i$ to obtain a homogeneous polynomial in the α_i of degree d, where d is the maximum of the d_i. Finally, we compute the coefficients $b_{\mathbf{k}}^d$, for $\mathbf{k} = (k_1, \ldots, k_n)$, $0 \leq k_i$, $\sum k_i = d$, in terms of the *generalized Bernstein base polynomials* $B_{\mathbf{k}}^d$. These generalized Bernstein base

polynomials are the terms in the expansion of

$$\begin{aligned} 1 &= (\alpha_1 + \alpha_2 + \cdots + \alpha_n)^d \\ &= \sum_{\substack{k_1,k_2,\ldots,k_n \geq 0 \\ k_1+k_2+\cdots+k_n=d}} \binom{d}{k_1, k_2, \ldots, k_n} \alpha_1^{k_1} \alpha_2^{k_2} \cdots \alpha_n^{k_n} = \sum_{\substack{k_1,k_2,\ldots,k_n \geq 0 \\ k_1+k_2+\cdots+k_n=d}} B_{\mathbf{k}}^d(\boldsymbol{\alpha}), \end{aligned}$$

where

$$\binom{d}{k_1, k_2, \ldots, k_n} = \frac{d!}{k_1! k_2! \ldots k_n!}$$

are the multinomial coefficients. Note that, again, the $B_{\mathbf{k}}^d(\boldsymbol{\alpha})$ are nonnegative and sum to 1 and so can be considered to be the coefficients in the expression of $p(\mathbf{x})$ as a convex combination of the $b_{\mathbf{k}}^d$. We therefore have

$$\min_{\substack{k_1,k_2,\ldots,k_n \geq 0 \\ k_1+k_2+\cdots+k_n=d}} b_{\mathbf{k}}^d \leq p(\mathbf{x}) \leq \max_{\substack{k_1,k_2,\ldots,k_n \geq 0 \\ k_1+k_2+\cdots+k_n=d}} b_{\mathbf{k}}^d \tag{6}$$

on the polytope $P \subset \mathbb{Q}^n$. The generalized Bernstein base polynomials we use here are different from the multivariate Bernstein polynomials [Zettler and Garloff 1998; Clauss and Tchoupaeva 2004], which are products of standard Bernstein polynomials.

Note that the algorithm outlined above does not require the points $\mathbf{v}_i$ to be the vertices of the polytope P. They may instead be any set of generators for the polytope P.

We may also consider *parametric polytopes* $P : D \to \mathbb{Q}^n : \mathbf{q} \mapsto P(\mathbf{q})$,

$$P(\mathbf{q}) = \left\{ \mathbf{x} \mid \exists \alpha_i \in \mathbb{Q} : \mathbf{x} = \sum_i \alpha_i \mathbf{v}_i(\mathbf{q}), \alpha_i \geq 0, \sum_i \alpha_i = 1 \right\},$$

where $D \subset \mathbb{Q}^r$ is the parameter domain and $\mathbf{v}_i(\mathbf{q}) \in \mathbb{Q}[\mathbf{q}]$ are arbitrary polynomials in the parameters $\mathbf{q}$. Note that some of these generators may be vertices for only a subset of the values of the parameters. The coefficients $a_{\mathbf{i}}$ of the polynomial $p(\mathbf{x})$ (5) may also themselves be polynomials in the parameters $\mathbf{q}$, i.e., $p(\mathbf{x}) \in (\mathbb{Q}[\mathbf{q}])[\mathbf{x}]$ and

$$a_{\mathbf{i}} = \sum_{j_1=0}^{m_1} \sum_{j_2=0}^{m_2} \cdots \sum_{j_r=0}^{m_r} b_{j_1,j_2,\ldots,j_r}\, q_1^{j_1} q_2^{j_2} \cdots q_r^{j_r}.$$

Applying the algorithm outlined above, we obtain parametric generalized Bernstein coefficients $b_{\mathbf{k}}^d(\mathbf{q})$ and parametric bounds

$$\min_{\substack{k_1,k_2,\ldots,k_n \geq 0 \\ k_1+k_2+\cdots+k_n=d}} b_{\mathbf{k}}^d(\mathbf{q}) \leq p(\mathbf{q})(\mathbf{x}) \leq \max_{\substack{k_1,k_2,\ldots,k_n \geq 0 \\ k_1+k_2+\cdots+k_n=d}} b_{\mathbf{k}}^d(\mathbf{q}).$$

The removal of redundant bounds in this expression is discussed in Section 3.1.

Example 2.3. Consider the polynomial $p(x_1, x_2) = \frac{1}{2}x_1^2 + \frac{1}{2}x_1 + x_2$ over the parametric polytope generated by the points $\binom{0}{0}$, $\binom{N}{0}$ and $\binom{N}{N}$. Hence any

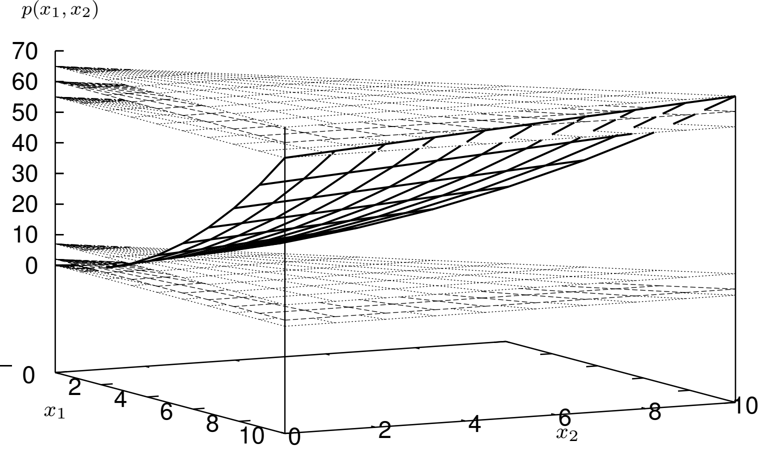

Fig. 2. The polynomial $p(x_1,x_2) = \frac{1}{2}x_1^2 + \frac{1}{2}x_1 + x_2$ and the corresponding Bernstein coefficients

point $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ in the polytope is a convex combination of these points:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \alpha_1 \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \alpha_2 \begin{pmatrix} N \\ 0 \end{pmatrix} + \alpha_3 \begin{pmatrix} N \\ N \end{pmatrix} \qquad 0 \leq \alpha_i \leq 1 \qquad \sum_{i=1}^{3} \alpha_i = 1$$

By replacing $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ with this convex combination, a new polynomial is obtained whose variables are $\alpha_1, \alpha_2, \alpha_3$:

$$\frac{1}{2}N^2\alpha_2^2 + N^2\alpha_2\alpha_3 + \frac{1}{2}N^2\alpha_3^2 + \frac{1}{2}N\alpha_2 + \frac{3}{2}N\alpha_3$$

Monomials of degree less than 2 are transformed into sums of monomials of degree 2:

$$\begin{aligned} \frac{1}{2}N\alpha_2 &= \frac{1}{2}N\alpha_2(\alpha_1 + \alpha_2 + \alpha_3) \\ \frac{3}{2}N\alpha_3 &= \frac{3}{2}N\alpha_3(\alpha_1 + \alpha_2 + \alpha_3). \end{aligned}$$

The final polynomial is:

$$\begin{aligned} p(\alpha_1, \alpha_2, \alpha_3) = & \left(\frac{1}{2}N^2 + \frac{1}{2}N\right)\alpha_2^2 + \left(\frac{1}{2}N^2 + \frac{3}{2}N\right)\alpha_3^2 \\ & + \frac{1}{2}N\alpha_1\alpha_2 + \frac{3}{2}N\alpha_1\alpha_3 + (N^2 + 2N)\alpha_2\alpha_3. \end{aligned}$$

The basis is built from the expansion of $(\alpha_1 + \alpha_2 + \alpha_3)^2$ providing the following

monomials:

$$
\begin{aligned}
B_{2,0,0} &= \alpha_1^2 \\
B_{0,2,0} &= \alpha_2^2 \\
B_{0,0,2} &= \alpha_3^2 \\
B_{1,1,0} &= 2\alpha_1\alpha_2 \\
B_{1,0,1} &= 2\alpha_1\alpha_3 \\
B_{0,1,1} &= 2\alpha_2\alpha_3.
\end{aligned}
$$

Rewriting $p(\alpha_1, \alpha_2, \alpha_3)$ in terms of this basis, we obtain

$$
0\,B_{2,0,0} + \left(\frac{1}{2}N^2 + \frac{1}{2}N\right) B_{0,2,0} + \left(\frac{1}{2}N^2 + \frac{3}{2}N\right) B_{0,0,2} + \frac{1}{4}NB_{1,1,0} + \frac{3}{4}NB_{1,0,1} + \left(\frac{1}{2}N^2 + N\right) B_{0,1,1}.
$$

It can then be concluded that the polynomial varies between 0 and $\frac{1}{2}N^2 + \frac{3}{2}N$. Since both of these coefficients are sharp coefficients, the bounds are exact bounds. The graph of the polynomial and the corresponding Bernstein coefficients are shown in Figure 2 for $N = 10$. □

3. COMMON OPERATIONS

In this section, we explain how to perform some operations that are common to many applications of Bernstein expansion. In particular, we provide more details on how to compute an upper bound of a polynomial over a parametric domain bounded by linear constraints and we show how to apply this computation to find a bound on the number of integer points in sets that can be described by linear constraints.

3.1 Bounding a Polynomial over a Parametric Domain

We already explained in Section 2.2 that given a polynomial and a set of parametric points, we can compute the Bernstein coefficients of the polynomial over the parametric convex polytope generated by the parametric points and that for any value of the parameters the minimum and maximum values over all Bernstein coefficients evaluated for this particular value of the parameters, provide a lower and an upper bound for the value of the polynomial over the convex polytope associated to these parameter values. However, in many situations where we wish to find a bound for a polynomial, the domain over which we wish to compute this bound is not given by a set of generators, but rather by a set of constraints. Also, when evaluating the lower or upper bound, we want to evaluate as few of the Bernstein coefficients as possible. We discuss these two issues in this section.

For example, suppose we want to compute an upper bound for the polynomial

$$
-\frac{1}{2}i^2 - \frac{3}{2}i - j - n^2 + 4n + 2in \tag{7}
$$

over the domain

$$
D(n) = \{\, (i,j) \mid 0 \le i \le 3n-1 \wedge 0 \le j \le n-1 \wedge 3n-1 \le i+j \le 4n-2 \,\}, \tag{8}
$$

where n is a parameter. The first step is to compute the (parametric) vertices of $D(n)$. If the domain is bounded by linear constraints in the variables and the parameters, then we can use `PolyLib` [Loechner 1999] to compute these vertices. The result is a subdivision of the parameter space in polyhedral cells, each with an associated set of parametric vertices [Clauss and Loechner 1998]. Note that we mentioned in Section 2.2 that the generators of a parametric polytope do not need to be vertices for all values of the parameters. However, they do obviously have to be inside the parametric polytope. Vertices associated with one subdomain that are not also associated with another subdomain will lie *outside* of this other subdomain. We therefore need to treat each subdomain separately. In the example, there is only one parameter domain and we find the vertices

$$\left\{ \begin{pmatrix} 2n \\ n-1 \end{pmatrix}, \begin{pmatrix} 3n-1 \\ 0 \end{pmatrix}, \begin{pmatrix} 3n-1 \\ n-1 \end{pmatrix} \right\} \qquad \text{if } n \geq 1.$$

If the constraints describing the domain are only linear in the variables (and not in the parameters), then we may still compute the vertices of the domain, but the subdomains of the parameter space that have a fixed set of parametric vertices will no longer be polyhedral [Rabl 2006].

The next step is to compute the Bernstein coefficients as explained in Section 2.2. For our example we obtain the coefficients

$$n^2 - \frac{n}{4} + \frac{5}{4}, \frac{n^2}{2} + \frac{n}{2} + 1, \frac{n^2}{2} + \frac{3}{2}, n^2 + 1, \frac{n^2}{2} - \frac{n}{2} + 2, n^2 - \frac{3n}{4} + \frac{7}{4}.$$

An upper bound $u(n)$ for the value of the polynomial over $D(n)$ is therefore

$$u(n) = \max\left\{ n^2 - \frac{n}{4} + \frac{5}{4}, \frac{n^2}{2} + \frac{n}{2} + 1, \frac{n^2}{2} + \frac{3}{2}, n^2 + 1, \frac{n^2}{2} - \frac{n}{2} + 2, n^2 - \frac{3n}{4} + \frac{7}{4} \right\} \quad \text{if } n \geq 1. \tag{9}$$

To compute the upper bound for any particular value of n, we therefore need to evaluate these 6 polynomials at this value and take the maximum. However, it is clear that some of these polynomials are redundant in the sense that for any value of the parameters in the domain the polynomial always evaluates to a smaller number than some other polynomial.

The simplest way to eliminate redundant Bernstein coefficients, is to examine the sign of the difference between two polynomials. If the sign is constant over the domain (where a zero sign may be treated as either positive or negative), then one of the two is redundant. Some easy ways of determining the sign of a (difference) polynomial are as follows.

- If the difference is a constant, the check is trivial.
- If the difference is linear in the parameters, we add the constraint that the difference be strictly larger than zero to the domain and check whether it becomes empty. For example, the polynomial $\frac{n^2}{2} + \frac{3}{2}$ is redundant since

$$\left(\frac{n^2}{2} + \frac{3}{2}\right) - \left(\frac{n^2}{2} + \frac{n}{2} + 1\right) = \frac{1}{2} - \frac{n}{2}$$

  and this difference polynomial is never greater than zero for $n \geq 1$. The polynomial $\frac{n^2}{2} - \frac{n}{2} + 2$ is eliminated for the same reason, while $n^2 - \frac{n}{4} + \frac{5}{4}$ and

$n^2 - \frac{3n}{4} + \frac{7}{4}$ are eliminated because they are redundant with respect to $n^2 + 1$. If it turns out that the sign of the difference varies over the domain, we could in principle further subdivide the domain along the above constraint.

—If the domain over which we want to determine the sign is bounded, we can apply Bernstein expansion again on the difference over this domain, which is now considered to be a fixed domain without parameters. The resulting Bernstein coefficients are therefore constants. If all the non-zero Bernstein coefficients have the same sign, then so will the difference over the whole domain. For example, if we assume that there is an upper bound on n, say 1000, then we can perform Bernstein expansion on

$$\left(\frac{n^2}{2} + \frac{n}{2} + 1\right) - \left(n^2 + 1\right) = -\frac{n^2}{2} + \frac{n}{2} \tag{10}$$

over $1 \leq n \leq 1000$. The resulting Bernstein coefficients are

$$\left\{0, -499500, \frac{-999}{4}\right\}$$

and so we can conclude that $\frac{n^2}{2} + \frac{n}{2} + 1$ is redundant with respect to $n^2 + 1$. Note that if the polynomial is univariate of degree d with coefficients c_i then we know that all real roots lie in the interval $[-M, M]$ with $M = 1 + \max_{0 \leq i \leq d-1} |c_i|/|c_d|$ (Cauchy's bound). It is therefore sufficient to consider the intersection of a strict superset of this interval with the possibly unbounded domain of interest. In the example, it would be sufficient to consider the domain $1 \leq n \leq 3$.

—If the domain over which we want to determine the sign is not bounded, but there is a lower bound on one of the parameters, we can write the Taylor expansion of the difference about this lower bound and determine the signs of the coefficients in the Taylor expansion. Note that we can easily compute these coefficients using synthetic division. If all signs are constant and equal, then also the difference will have this constant sign. For example, we can write (10) as

$$-\frac{1}{2}(n-1) - \frac{1}{2}(n-1)^2$$

and the coefficients are clearly negative, so we can again conclude that $\frac{n^2}{2} + \frac{n}{2}$ is redundant, over the whole domain $n \geq 1$.

In our example we have now been able to simplify (9) to

$$u(n) = n^2 + 1 \qquad \text{if } n \geq 1. \tag{11}$$

In general, we will however not be able to identify all but one polynomial as redundant. Still, it may be desirable in some cases to have only a single polynomial associated to every subdomain, such that for a given subdomain only this single polynomial needs to be evaluated. If the difference between two polynomials is linear then this can easily be accomplished by splitting the domain along the hyperplane where the difference is zero. For example, suppose we have two polynomials $n^2 + 3n - 500$ and $n^2 + n$ in the maximum expression associated to the domain $n \geq 4$. The difference between these two polynomials $2n - 500$ is zero along $n = 250$ and so we would split the domain into, say, $4 \leq n < 250$ and $250 \leq n$. If

```
1 for (i = 0; i < 4*n-1; ++i)
      for (j = 0; j < n; ++j) {
          if (i+j >= n-1 && i+j <= 3*n-2)
              a[i][j] = f(i,j);
          if (i+j >= 2*n-1 && i+j <= 4*n-2)
6             b[j] = b[j] + a[i-n][j];
      }
```

Fig. 3. A nested loop with temporary array a

the differences between pairs of polynomials is not linear, but they are univariate, then we may not be able to easily split the domain into subdomains where only a single polynomial remains, but based on Cauchy's bounds, we can identify and split off a region of "big" values where the upper bound is given by a single polynomial.

3.2 Bounding the Number of Elements in a Set

A common problem in compiler analysis is that of finding a bound on the number of elements in a set of integers. In this section we describe how to use Bernstein expansion to solve this problem in the case the set is described by linear constraints. A typical example of such an analysis is that 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.

Consider, for example, the code fragment in Figure 3 and assume that array a is a temporary array only used inside this loop nest. Let us concentrate on the subproblem of finding the maximal number of live elements in the array a. For each iteration of the loop nest, we can describe the set of elements of the array a that have already been defined and that still need to be used and we want to compute the maximal number of elements in this set over all iterations of the loop nest. In this simple example, each array element that is defined in line 4 is used exactly once in line 6 and is therefore live between its definition and its first and only use. The number of elements live at a given iteration is therefore simply the number of elements defined before that iteration minus the number of elements used before that iteration, i.e.,

$$L(n,i,j) = |\{(i',j') \in D_1(n) \mid (i',j') \preccurlyeq (i,j)\}| - |\{(i',j') \in D_2(n) \mid (i',j') \prec (i,j)\}|, \tag{12}$$

with

$$D_1(n) = \{(i,j) \mid 0 \le i < 4n-1 \wedge 0 \le j < n \wedge n-1 \le i+j \le 3n-2\}$$

and

$$D_2(n) = \{(i,j) \mid 0 \le i < 4n-1 \wedge 0 \le j < n \wedge 2n-1 \le i+j \le 4n-2\}$$

the iteration domains of the definition and the use respectively, and $\prec$ denoting "lexicographically smaller than". Note that $(i',j') \prec (i,j) \equiv i' < i \vee (i' = i \wedge j' < j)$ is not a linear constraint, but rather a disjunction of linear constraints. If we want to describe our sets using only (conjunctions of) linear constraints, we will therefore

need to split each of the counting problems in (12) into two separate counting problems. The maximal number of live elements is then simply

$$M(n) = \max_{(i,j)\in D_2(n)} L(n,i,j).$$

Note that it is sufficient to compute the maximum of $D_2(n)$ rather than the whole iteration domain since the maximal number of live elements will always occur right before an element is used.

In general we see that we can identify three sets of variables in such problems:

—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.

Counting the number of integer elements in a parametric set bounded by linear constraints can be performed very efficiently using Barvinok's algorithm [Barvinok and Pommersheim 1999; Verdoolaege et al. 2007]. If the description also contains existentially quantified variables, then some extensions can be used that work fairly well in practice [Verdoolaege et al. 2005; Seghir and Loechner 2006]. The result of this counting problem is a piecewise quasi-polynomial, i.e., a subdivision of the parameter space (of the counting problem), with a quasi-polynomial associated to each region of the subdivision. These piecewise quasi-polynomials that result from counting problems are also called Ehrhart polynomial by some authors [Clauss and Loechner 1998]. A quasi-polynomial is a polynomial expression where the coefficients depend periodically on the variables. We can, however, avoid this periodicity (i.e., obtain an actual polynomial) by approximating the original parametric polytope to obtain either an underestimate or an overestimate [Meister 2004]. In our example, the number of elements defined before an iteration of $D_2(n)$ is

$$\begin{cases} ni + j - \frac{1}{2}n^2 + \frac{1}{2}n + 1 & \text{if } (i,j) \in D_2(n) \wedge i \le 2n-1 \\ -\frac{1}{2}i^2 + 4ni - \frac{1}{2}i + j - \frac{5}{2}n^2 + \frac{3}{2}n + 1 & \text{if } (i,j) \in D_2(n) \wedge i \ge 2n \wedge i + j \le 3n-2 \\ -\frac{1}{2}i^2 + 3ni - \frac{3}{2}i - \frac{5}{2}n^2 + \frac{9}{2}n & \text{if } (i,j) \in D_2(n) \wedge i + j \ge 3n-2 \wedge i \le 3n-1 \\ 2n^2 + 1 & \text{if } (i,j) \in D_2(n) \wedge i \ge 3n, \end{cases}$$

while the number of elements used before an iteration of $D_2(n)$ is

$$\begin{cases} \frac{1}{2}i^2 - ni + \frac{3}{2}i + j + \frac{1}{2}n^2 - \frac{5}{2}n + 1 & \text{if } (i,j) \in D_2(n) \wedge i \le 2n-1 \\ ni + j - \frac{3}{2}n^2 + \frac{1}{2}n & \text{if } (i,j) \in D_2(n) \wedge i \ge 2n \wedge i + j \le 3n-2 \\ ni + j - \frac{3}{2}n^2 + \frac{1}{2}n & \text{if } (i,j) \in D_2(n) \wedge i + j \ge 3n-2 \wedge i \le 3n-1 \\ -\frac{1}{2}i^2 + 4ni - \frac{1}{2}i + j - 6n^2 + 2n & \text{if } (i,j) \in D_2(n) \wedge i \ge 3n. \end{cases}$$

The number of live elements $L(n,i,j)$ at a given iteration of $D_2(n)$ is therefore

$$\begin{cases} 2ni - n^2 + 3n - \frac{1}{2}i^2 - \frac{3}{2}i & \text{if } (i,j) \in D_2(n) \wedge i \le 2n-1 \\ -\frac{1}{2}i^2 + 2ni - \frac{1}{2}i - n^2 + n + 1 & \text{if } (i,j) \in D_2(n) \wedge i \ge 2n \wedge i + j \le 3n-2 \\ -\frac{1}{2}i^2 + 2ni - \frac{3}{2}i - n^2 + 4n - j & \text{if } (i,j) \in D_2(n) \wedge i + j \ge 3n-2 \wedge i \le 3n-1 \\ 8n^2 + \frac{1}{2}i^2 - 4ni + \frac{1}{2}i - j - 2n + 1 & \text{if } (i,j) \in D_2(n) \wedge i \ge 3n. \end{cases}$$

We now proceed as in Section 3.1 on each of the subdomains in the result of the counting problem. That is, for each subdomain, we compute its vertices and then compute the Bernstein coefficients for each of the subdomains in the parameter space with a fixed set of vertices. In our example, each subdomain of the parameter space of the counting problem yields a single subdomain of the parameter space of the maximization problem. The domain $D_2(n) \cap \{(i, j) \mid i + j \geq 3n - 2 \wedge i \leq 3n - 1\}$ has already been handled in Section 3.1. For the other domains we obtain for the set $D_2(n) \cap \{(i, j) \mid i \leq 2n - 1\}$, the vertices

$$\left\{ \begin{pmatrix} 2n-1 \\ n-1 \end{pmatrix}, \begin{pmatrix} n \\ n-1 \end{pmatrix}, \begin{pmatrix} 2n-1 \\ 0 \end{pmatrix} \right\} \quad \text{if } n \geq 1,$$

for the set $D_2(n) \cap \{(i, j) \mid i \geq 2n \wedge i + j \leq 3n - 2\}$, the vertices

$$\left\{ \begin{pmatrix} 3n-2 \\ 0 \end{pmatrix}, \begin{pmatrix} 2n \\ 0 \end{pmatrix}, \begin{pmatrix} 2n \\ n-2 \end{pmatrix} \right\} \quad \text{if } n \geq 2,$$

and for the set $D_2(n) \cap \{(i, j) \mid i \geq 3n\}$, the vertices

$$\left\{ \begin{pmatrix} 4n-2 \\ 0 \end{pmatrix}, \begin{pmatrix} 3n \\ 0 \end{pmatrix}, \begin{pmatrix} 3n \\ n-2 \end{pmatrix} \right\} \quad \text{if } n \geq 2.$$

On the first domain, the Bernstein coefficients are

$$\left\{ n^2 + \frac{n}{4} + \frac{3}{4}, n^2 + 1, \frac{n^2}{2} + \frac{3}{2}n \right\},$$

on the second domain, the Bernstein coefficients are

$$\left\{ n^2 + 1, n^2 - \frac{n}{4} + \frac{3}{2}, \frac{n^2}{2} + \frac{3}{2}n \right\}.$$

while on the final domain, the Bernstein coefficients are

$$\left\{ 2, \frac{n^2}{2} - \frac{3}{2}n + 3, \frac{n^2}{2} - n + 2, \frac{3}{4}n + \frac{1}{2}, \frac{n^2}{2} - \frac{n}{2} + 1, \frac{n}{4} + \frac{3}{2} \right\}.$$

The maximum of all these coefficients, including those from Section 3.1, is therefore an upper bound on the number of live elements for $n \geq 2$.

In general we obtain for each subdomain of the counting problem a subdivision of the parameter space with a set of Bernstein coefficients associated to each cell. The bounds on the set are then given by the common refinement of the parameter space subdivisions over all maximization problems where the set of Bernstein coefficients associated to each cell in the common refinement is the union of the sets of Bernstein coefficients associated to the corresponding cells in the individual solutions. In our example, the common refinement consists of two cells: $n = 1$, with coefficients from two problems only, and $n \geq 2$, with coefficients from all problems. In the first cell, we can evaluate the polynomials and the upper bound is simply 1, while in the second cell we can remove redundant polynomials as described in Section 3.1 and the only remaining polynomial is $n^2 + \frac{n}{4} + \frac{3}{4}$. Our upper bound on the number of

live elements is therefore

$$\begin{cases} 2 & \text{if } n = 1 \\ n^2 + \frac{n}{4} + \frac{3}{4} & \text{if } n \geq 2, \end{cases}$$

which can be simplified to $n^2 + \frac{n}{4} + \frac{3}{4}$ if $n \geq 1$.

4. SOFTWARE IMPLEMENTATION

We have implemented the computation of a bound on an arbitrary multivariate polynomial defined over a linearly parameterized convex polytope, as explained in Section 3.1, in our `bernstein` library. This includes the computation of the Bernstein coefficients of the polynomial as well as the removal of redundant coefficients. Our library is built on top of two other libraries:

—the polyhedral library `PolyLib` [Loechner 1999] to compute the vertices of a linearly parameterized polytope,

—the `GiNaC` library [Bauer et al. 2002] for symbolic polynomial manipulations.

Both of these libraries use the GMP library [GMP] (as part of the `CLN` [Haible 2006] library in the case of `GiNaC`) for arbitrary precision arithmetic on integers. Furthermore, the `bernstein` library has been integrated into the `barvinok` library [Verdoolaege 2006], which has been augmented with a procedure for computing a bound on the result of a counting problem using `bernstein`. This procedure effectively implements the approach discussed in Section 3.2.

5. APPLICATIONS TO MEMORY REQUIREMENT ESTIMATION

In this section, we describe some applications to memory requirement estimation. In each of these applications we are given a polynomial expression of the amount of "memory in use" at a given "execution point" and we want to compute an upper bound on the amount of memory used over all execution points. The memory in use can be the set of live array elements, the tokens in a FIFO, the elements accessed between two uses of the same element or the size of the memory scope of a method in terms of its parameter values. Our technique can also be used to extend the applicability of the applications of Clauss and Tchoupaeva [2004] from "boxes" to parametric polytopes. We will not repeat those applications here.

5.1 Memory Size Computation

The problem of computing the "exact memory size" of a program is that of finding the minimum amount of memory locations needed to store the data of the program during its execution [Zhu et al. 2006]. This problem is basically the liveness analysis we used as an example in Section 3.2 and variations of this problem have been studied earlier in the literature (e.g., [Verbauwhede et al. 1994; Grun et al. 1998; Zhao and Malik 2000; Ramanujam et al. 2001]). Zhu et al. [2006] distinguish themselves from previous research by computing the memory size exactly, rather than approximately. We focus on their work because it is the most recent and because they cite some numbers to which we can compare our results. They propose a rather complicated algorithm where they first decompose the array references

into disjoint linearly bounded lattices and then compute the number of live elements both between consecutive top-level loops and inside top-level loops. For this last computation, they determine the iteration where the number of live elements changes and then presumably iterate over all these iterations to find the maximum number of live elements. Their algorithm is fundamentally non-parametric, so they need to redo the whole computation for each value of the parameters.

Using Bernstein expansion, we can compute (an upper bound of) the memory size parametrically. We implemented a very straightforward algorithm where we first compute pairs of consecutive accesses to array elements, where the first is either a write or a read and the second is a read. We perform this computation using a variation of array dataflow analysis [Feautrier 1991], resulting in a union of relations described by linear constraints. For each statement in the program and for each relation in this union, we then compute the number of live elements in the relation as a function of the iterators of the enclosing loops and sum these together for all relations in the union. The number of live elements is determined by projecting the relation on both the first and the second access, computing the number of both these accesses that precede a given iteration of the statement using `barvinok` [Verdoolaege 2006] and taking the difference. The maximum number of live elements is then computed as explained in Section 3.2.

Although the procedure outlined above can still be significantly optimized by avoiding redundant computations, even the straightforward implementation can compute the parametric memory size for the 2D Gaussian blur filter in 44 seconds on an Athlon MP 1500+ with 512MiB internal memory, while Zhu [2006] reports computation times of 3 and 103 seconds on a slightly faster machine for parameter values $N = 100$, $M = 50$ and $M = N = 500$ respectively. The size we compute is $MN + 5$, which agrees with the values 5005 and 250005 reported by Zhu [2006]. We should point out that the algorithm of Zhu et al. [2006] appears to be fairly inefficient for large values of the parameters. In an alternative, again very straightforward, implementation, we first basically perform the dependence analysis outlined above and generate code using `CLooG` [Bastoul 2004] to count the number of live elements by incrementing a counter each time a value is read or written that is still needed and decrementing the same counter each time a value is read and report the maximal value attained by the counter. For each value of the parameters we then compile and execute the generated code. For the same application, we found that the analysis and code generation takes about 9 seconds, compilation takes about 0.5 seconds and the actual execution is too fast to be measured for $N = 100$, $M = 50$ while it takes about 0.03 seconds for $M = N = 500$.

Note that the size computed by our procedure may be an overestimate. However, the actual memory size may not be very useful, since in order to fit all data in the "exact memory size" you would still have to derive an appropriate mapping of the array elements to this minimally sized memory. This addressing issue is not discussed by Zhu et al. [2006].

5.2 Computing FIFO Sizes in Process Networks

The conversion of a sequential program to a process network is a way of exposing the task-level parallelism in the program [Turjan et al. 2004; Verdoolaege et al. 2006]. In a process network, independent processes communicate with each other

through communication channels. The derivation of process networks is an extension of array dataflow analysis [Feautrier 1991], where array reads are analyzed to determine where the data was produced and where all array accesses are subsequently replaced by reads and writes to the communication channels. In many cases, the reads and writes occur (or can be made to occur) in the same order and the communication channel is a FIFO. In an idealized form, these FIFOs are unbounded, but for a practical (hardware or software) implementation we need to be able to compute bounds on the sizes of the FIFOs.

We first consider self-loops, i.e., FIFOs from a given process to itself. The iteration order inside any given process is fixed and corresponds to the iteration order in the original program. To compute the maximal number of tokens in the FIFO, we again apply the technique of Section 3.2. The set for which we want to compute an upper bound is the set of tokens in the FIFO for any iteration of the process and it is composed of the tokens that have been written to the FIFO but have not been read yet. To count the number of elements in this set, we count the number of *write* operations that precede the given iteration as well as the number of *read* operations that precede the iteration and take the difference.

For FIFOs between two distinct processes, it is in general impossible to know how many tokens are in the FIFO at any given instant of time because the processes are essentially independent. The only influence they exert on each other is through the communication channels. If a process reads from an empty FIFO, it will block until data is available. Likewise, if a process writes to a full FIFO, it will block until sufficient room is available. Note that if the size of a FIFO is too small, then the network will deadlock. FIFO sizes that are too large, however, waste resources. Our objective is therefore to find the smallest FIFO sizes that still ensure a deadlock free execution. Note that in practice we would not necessarily use the absolute smallest sizes, since they could hinder the parallel execution of the processes as these could spend a substantial amount of time blocking on reads or writes. Knowledge of the smallest sizes is however a good starting point for finding good sizes.

Unfortunately, computing the minimal deadlock-free FIFO sizes is a non-trivial global optimization problem. The easiest way to obtain (non-minimal) deadlock-free FIFO sizes is to take the declared sizes of the arrays whose elements are sent across the FIFOs, but this typically results in a huge overestimate. Another option is to take the schedule of the original sequential program and compute the FIFO sizes as for self-loops, but this may again lead to a substantial overestimate. To improve on this estimate we instead first compute a global schedule, independent of the schedule of the sequential program, that strives to minimize the FIFO sizes [Verdoolaege et al. 2006].

Our approach greedily combines iteration domains of different statements until all iteration domains share a common iteration space. The relative position of two iteration domains that are combined together is chosen such that the minimal distance vector of any dependence between the two iteration domains is zero, meaning that at least one token is used immediately after it has been produced. This algorithm ensures that a valid schedule is found, provided that it starts from a sequential program [Verdoolaege et al. 2003].

Example 5.1. Consider the program in Figure 4. The process network derived

```
for (j = 0; j < N; ++j)
    for (i = j; i < N; ++i)
        R[j][i] = Zero();
for (k = 0; k < K; ++k)
    for (j = 0; j < N; ++j)
        X[k][j] = ReadMatrix();
for (k = 0; k < K; ++k)
    for (j = 0; j < N; ++j) {
        Vectorize(R[j][j], X[k][j], &R[j][j], &X[k][j], &t);
        for (i = j+1; i < N; ++i)
            Rotate(R[j][i], X[k][i], t, &R[j][i], &X[k][i], &t);
    }
for (j = 0; j < N; ++j)
    for (i = j; i < N; ++i)
        WriteMatrix(R[j][i]);
```

Fig. 4. QR algorithm

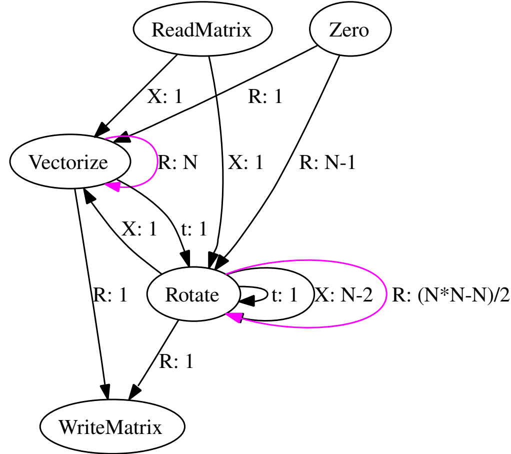

Fig. 5. QR Process Network

from this code using the methods of Verdoolaege et al. [2006] is shown in Figure 5. The network consists of 5 nodes, corresponding to the statements in the original program, and 12 edges, 4 of which are self-loops. For example, the edge from the `Vectorize` node to itself has the following dependence relation, relating iterations that write to the FIFO to the corresponding iteration that reads from the FIFO:

$$\{[k, j] \to [k', j'] : k' = 1 + k \wedge j' = j \wedge 0 \le j < N \wedge 0 \le k \le K - 2\}.$$

Projection on domain and range yields

$$W = \{[k, j] : 0 \le j \le N - 1 \wedge 0 \le k \le K - 2\}$$

and

$$R = \{[k, j] : 0 \le j \le N - 1 \wedge 1 \le k \le K - 1\}$$

```
for (a = 0; a <= K-1 a++)
  for (b = 0; b <= N-1; b++)
    for (c = b; c <= 2*N-2; c++) {
      if (a == 0 && c <= N-1)
        R[b][c] = Zero();
      if (b == 0 && N-1 <= c)
        X[a][c-N+1] = ReadMatrix();
      if (c == N-1)
        Vectorize(R[b][b], X[a][b], &R[b][b], &X[a][b], &t);
      if (b <= c-N)
        Rotate(R[b][c-N+1],X[a][c-N+1],t,&R[b][c-N+1],&X[a][c-N+1],&t);
      if (a == K-1 & b <= c-N+1)
        WriteMatrix(R[b][c-N+1]);
    }
```

Fig. 6. Rescheduled QR algorithm

for the sets of iterations that write to and read from the FIFO respectively. The number of elements of W that precede a given iteration (k, j) in R is

$$\begin{cases} Nk & \text{if } 0 \leq j \leq N-1 \wedge k = K-1 \\ Nk + j & \text{if } 0 \leq j \leq N-1 \wedge 1 \leq k \leq K-2, \end{cases}$$

while the number of elements of R that precede the same iteration is $Nk + j - N$. The number of elements in the FIFO at iteration (k, j) is therefore

$$\begin{cases} N - j & \text{if } 0 \leq j \leq N-1 \wedge k = K-1 \\ N & \text{if } 0 \leq j \leq N-1 \wedge 1 \leq k \leq K-2 \end{cases}$$

and the maximum is clearly N.

As explained above, to compute deadlock-free FIFO sizes for the other edges, we place the iteration domains in a common space. In order to be able to do so, however, we first need to ensure all iteration domains have the same dimension. We apply a simple heuristic that inserts fixed-valued iterators in the lower dimensional iteration domains based on the dependences they share with higher dimensional iteration domains. For example, the dependence between the `Rotate` and `WriteMatrix` nodes is described by

$$\{[k, j, i] \to [j', i'] : k = K-1 \wedge j' = j \wedge i' = i \wedge 0 \leq j < i < N\}.$$

Since both j' and j as well as i' and i are equal to each other up to a constant (in this case 0), inserting an extra iterator with an arbitrary value in the first position will allow a relative offset to be chosen that places the write and read on top of each other. Similarly, an extra iterator is inserted in the first position for node `Zero`, in the second position for node `ReadMatrix` and in the third position for node `Vectorize`. The algorithm then greedily combines the iteration domains into a common iteration space, in each step choosing a relative offset that minimizes some distance vectors. Finally, an extra iterator is added in the last position to ensure that all reads occur after the corresponding writes. For example, the above dependence between the `Rotate` and `WriteMatrix` nodes is now described by

$$\{[a, b, c, 1] \to [a', b', c', 2] : a' = a = K-1 \wedge b' = b \wedge c' = c \wedge N \leq c \leq 2N-2 \wedge 0 \leq b \leq c-N\},$$

where the innermost iterator takes the values 1 and 2 respectively. Writing out the (re)scheduled algorithm would result in the code shown in Figure 6. In the common iteration space, the computation of the FIFO sizes can proceed as for self-loops. The final results are shown in Figure 5.

5.3 Reuse Distances

The (forward) reuse distance of a memory access to a memory element is the number of distinct memory elements accessed between the given access and the next access to the same memory element. It is a measure for the locality of the access as the element will still be in the cache on the next access depending on whether the reuse distance is smaller than the cache size, assuming that the cache is fully associative with LRU replacement policy [Beyls and D'Hollander 2005]. Beyls and D'Hollander [2005] propose to use the reuse distance to select cache hints. Since the cache hint of a given instruction is fixed during the entire execution of the program, while it may give rise to many accesses with different reuse distances, they propose to base their cache hint selection on the cache that is sufficiently large to hold 90% of the elements accessed by the instruction until their next use.

To be able to determine the appropriate cache size, they need to *evaluate* the reuse distance for each loop iteration of the loops surrounding the given instruction, even though the reuse distance itself can be computed parametrically in terms of the loop iterators and structural parameters. Although Bernstein expansion cannot help to easily determine the minimum size that will hold 90% of the accesses, it can help to determine the minimum size that will hold all accesses that will still be reused, by computing an upper bound of the reuse distance over all iterations. This strategy can be further refined by also considering the lower bound of the reuse distance, which can be computed in a similar way or simply by noticing that $\min f(\mathbf{i}) = -\max -f(\mathbf{i})$, as well as the average reuse distance, which can also be computed parametrically [Verdoolaege 2005, Section 4.5.4].

5.4 Estimating Dynamic Memory Requirements

Braberman et al. [2006] present a static analysis approach for computing a parametric upper-bound of the amount of memory dynamically allocated by (Java-like) imperative object-oriented programs. Their major contribution is a technique to quantify dynamic allocations performed by a method. Given a method `m` with parameters $p_1, \ldots, p_k$, they propose an algorithm that computes a parametric polynomial in $p_1, \ldots, p_k$ that over-approximates the amount of memory allocated during the execution of `m`, i.e., all the dynamic memory claimed from the memory manager without considering any kind of (garbage) collection.

Roughly speaking, the technique works as follows: For every allocation statement, an invariant is derived that relates program variables in such a way that the amount of consumed memory is a function of the number of integer points that satisfy the invariant. This number is given in a parametric form as a polynomial where the unknowns are method parameters. The technique does not require annotating the program in any form and produces polynomials that bound the memory usage.

Combining this algorithm with static pointer and escape analysis [Salcianu and Rinard 2001; Blanchet 1999], it is also possible to compute memory region sizes to be used in scope-based memory management [Garbervetsky et al. 2004; Cherem

```
void m0(int m) {
  for (c = 0; 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();
  return arrB;
}
```

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

Fig. 7. Dynamic memory allocation example

and Rugina 2004]. In a scope-based memory management the heap is basically divided into regions associated with computation units (methods or threads). Escape analysis is used to decide in which region objects have to be allocated. This kind of memory management strategy is often used as an alternative to garbage-collected memory management in environments where performance or predictability are required. In general, the developer has to provide upper bounds of the size of each region, in order to ensure these performance requirements.

Given a method `m` with parameters $p_1, \ldots, p_k$, the paper presents algorithms for computing parametric polynomials in $p_1, \ldots, p_k$ that over-approximate the amount of memory that *escapes from* and is *captured by* `m` respectively.[1] In particular, the latter can be used as an upper bound for the size of the region associated to `m`. In this section, we will present a simplified version of this algorithm.

In our simplified algorithm, we will assume that memory only escapes from a method as a result of returning a reference to the memory to the calling method and that no method is called recursively. The required memory size of the region associated to a given method `m`, $\text{memRq}_{\mathtt{m}}$, is then the amount of memory captured by the method, $\text{cap}_{\mathtt{m}}$, plus the maximum of the memory sizes of the regions associated to all methods called by `m`, i.e.,

$$\text{memRq}_{\mathtt{m}} = \text{cap}_{\mathtt{m}} + \max_{\mathtt{p} \text{ called by } \mathtt{m}} \text{memRq}_{\mathtt{p}}.$$

Note that both $\text{cap}_{\mathtt{m}}$ and $\text{memRq}_{\mathtt{m}}$ will depend on the values of the parameters of `m` and so the maximum in the formula above needs to be taken over all invocations of all methods called by `m`. If a method is called from within a loop nest with affine loop bounds, then the maximum (or at least an upper bound) of the sizes of all invocations of that method in the loop nest can be computed using Bernstein expansion as explained in Section 3.1. The remaining maximum expression can be simplified in the same way as redundant Bernstein coefficients are eliminated in the same section.

Consider, for example, the code in Figure 7 and assume for simplicity that all objects are of size 1. We will use $\text{ret}_{\mathtt{m}}$ to denote the size of the memory returned from

[1] An object escapes a method `m` when its lifetime is longer that the lifetime of `m`. An object is captured by the method `m` when it can be safely collected at the end of the execution of `m`.

a method `m`. Method `m2` does not call any other methods, captures all allocations assigned to `b` and returns `arrB`. We therefore have

$$\begin{aligned}
\mathrm{cap}_{\mathtt{m2}}(n) &= \sum_{1\le j\le n} 1 = n\\
\mathrm{ret}_{\mathtt{m2}}(n) &= n\\
\mathrm{memRq}_{\mathtt{m2}}(n) &= \mathrm{cap}_{\mathtt{m2}}(n) = n.
\end{aligned}$$

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`. We have

$$\begin{aligned}
\mathrm{cap}_{\mathtt{m1}}(k) &= \sum_{1\le i\le k} (1+\mathrm{ret}_{\mathtt{m2}}(i)) = \sum_{1\le i\le k}(1+i) = \frac{k(k+3)}{2}\\
\mathrm{memRq}_{\mathtt{m1}}(k) &= \mathrm{cap}_{\mathtt{m1}}(k) + \max_{1\le i\le k} \mathrm{memRq}_{\mathtt{m2}}(i) = \frac{k(k+3)}{2} + k = \frac{k^2+5k}{2},
\end{aligned}$$

where Bernstein expansion is used to compute $\max_{1\le i\le k} \mathrm{memRq}_{\mathtt{m2}}(i)$ and we again assume that $k \ge 0$. Finally, for method `m0`, we have

$$\begin{aligned}
\mathrm{cap}_{\mathtt{m0}}(m) &= \sum_{0\le c\le m-1} \mathrm{ret}_{\mathtt{m2}}(2m-c) = \frac{3m^2+m}{2}\\
\mathrm{memRq}_{\mathtt{m0}}(m) &= \mathrm{cap}_{\mathtt{m0}}(m) + \max\left(\max_{0\le c\le m-1} \mathrm{memRq}_{\mathtt{m1}}(c), \max_{0\le c\le m-1} \mathrm{memRq}_{\mathtt{m2}}(2m-c)\right)\\
&= \mathrm{cap}_{\mathtt{m0}}(m) + \max\left(\max_{0\le c\le m-1} \frac{c^2+5c}{2}, \max_{0\le c\le m-1}(2m-c)\right)\\
&= \frac{3m^2+m}{2} + \max\left(\frac{m^2+3m-4}{2}, 2m\right)\\
&= \frac{3m^2+m}{2} + \frac{m^2+3m-4}{2}\\
&= 2m^2+2m-2,
\end{aligned}$$

where we now assume that $m \ge 1$. The general solution is

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

Note that in general, the solution may still contain maximum expressions over a finite set of polynomials, which will need to be evaluated at run-time when the values of the parameters are known. This computation can be performed fairly efficiently using the techniques of, e.g., Hosangadi et al. [2006]. Also note that our simplified algorithm works bottom-up, starting from the leaves in the call graph (which is assumed to be acyclic) and working its way up to the root. This process may lead to loss of precision. The actual algorithm therefore proceeds in a top-down fashion, computing the size required for an (indirectly) called method for each path in the call graph leading to that method. For more information we refer to Fernández [2006].

6. RELATED WORK

6.1 Handling Polynomials in Program Analysis

Maslov and Pugh [1994] present a technique to simplify polynomial constraints. It is based on a decomposition of any polynomial constraint into a conjunction of affine constraints and 2-variable hyperbolic and elliptical inequalities and equalities that can later be linearized. Hence their approach is not general and can only handle those polynomials that can be decomposed in this way.

Blume and Eigenmann [1994] present an algorithm for determining the sign of a symbolic expression. It is assumed that each variable has a (symbolic) lower and upper bound and these ranges are repeatedly substituted in the expression until a non-negative constant lower bound or a non-positive constant upper bound is found on the expression. 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 they can determine, by recursively applying their algorithm to the first order forward difference of the polynomial, that the expression is monotonically non-increasing or non-decreasing in a given variable, then they can safely substitute the lower and upper bounds of the variable simultaneously in the whole expression, leading to a tighter bound. Although this technique was only intended for determining the sign of a symbolic expression, it can also be used to find a symbolic bound by simply not substituting some of the variables. The main disadvantages of this technique are that it only works over "boxes" and that the accuracy can be very low for non-monotonic expressions. Applying the basic substitution technique to the example polynomial (7) from Section 3.1 over the box $[2n : 3n-1] \times [0 : n-1]$ yields an upper bound of $3n^2-n-1$. Exploiting the monotonicity of the example polynomial (over the edges of the box; the polynomial is not monotonic over the edges of the polytope D (8)) results in the upper bound n^2+n-1. This should be compared with the upper bound n^2+1 (11) obtained through Bernstein expansion.

Van Engelen et al. [2003] apply the same monotonicity test of Blume and Eigenmann [1994] in the specific context of bounding a polynomial over the parameterized box $[0, n-1]$. They may not have been aware of this earlier result since in their related work they refer to the corresponding conference paper [Blume and Eigenmann 1995] which lacks a description of the monotonicity test. They compute the forward difference in a slightly different way, though. In particular, they use the Newton series representation of a polynomial, which expresses the polynomial in the "falling factorial" basis. Although this representation is very useful for computing sums of polynomials, as shown in the same paper, it is not immediately obvious why it would be advantageous to use for computing the forward difference.

Fahringer [1998] describes an extension of the technique of Blume and Eigenmann [1994] for determining the sign of a symbolic expression to handle multiple lower and upper bounds on a variable. The semantics of his intermediate expressions are not clearly defined, however. It is therefore not clear whether his technique can also be used to find symbolic bounds on expressions.

Bernstein expansion for arbitrary intervals and multivariate polynomials was used in a previous work dealing with symbolic Bernstein expansion [Clauss and Tchoupaeva 2004]. However, this technique has some drawbacks that are described

in the introduction of this paper and that are entirely removed with the method presented in this paper.

6.2 Memory Requirement Estimation

6.2.1 *Static Memory.* The problem of finding (an approximation) of the minimal amount of memory required to run a program given a schedule has been studied before [Verbauwhede et al. 1994; Grun et al. 1998; Zhao and Malik 2000; Ramanujam et al. 2001; Zhu et al. 2006]. However, most authors make simplifying assumptions, such as uniformly generated references [Ramanujam et al. 2001], or the assumption that the number of live elements is an affine expression of the iterators [Zhao and Malik 2000], rather than the more general case of a polynomial. The techniques of other authors [Zhu et al. 2006] only work for non-parametric programs. Kjeldsberg et al. [2004] estimate the memory requirements when the execution order is only partially known. Many authors have also considered the problem of finding good memory mappings. We refer to Darte et al. [2005] for an overview and a mathematical framework for handling this problem.

6.2.2 *Dynamic Memory.* The problem of dynamic memory estimation has been studied for functional languages by Hofmann and Jost [2003], Hughes and Pareto [1999] and Unnikrishnan et al. [2003]. The work of Hofmann and Jost [2003] statically infers, by type derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs running under a special memory mechanism (free list of cells and explicit deallocation in pattern matching). The computed expressions are linear constraints on the sizes of various parts of data. Hughes and Pareto [1999] propose a variant of ML together with a type system based on the notion of sized types [Hughes et al. 1996], such that well typed programs are proven to execute within the given memory bounds. The technique proposed by Unnikrishnan et al. [2003] consists in, given a function, constructing a new function that symbolically mimics the memory allocations of the former. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. The evaluation of the bound function might not terminate, even if the original program does.

For imperative object-oriented languages, solutions have been proposed by Gheorghioiu [2002] and Chin et al. [2005]. The technique of Gheorghioiu [2002] manipulates symbolic arithmetic expressions on unknowns that are not necessarily program variables, but added by the analysis to represent, for instance, loop iterations. The resulting formula has to be evaluated on an instantiation of the remaining unknowns to obtain the upper-bound. No benchmarking is available to assess the impact of this technique in practice. Nevertheless, two points may be made. Since the unknowns may not be program inputs, it is not clear how instances are produced. Second, it seems to be quite over-pessimistic for programs with dynamically created arrays whose size depends on loop variables. The method proposed by Chin et al. [2005] relies on a type system and type annotations, similar to Hughes and Pareto [1999]. It does not actually synthesize memory bounds, but statically checks whether size annotations (Presburger formulas) are verified. It is therefore up to the programmer to state the size constraints, which are moreover required to be linear.

7. CONCLUSION

Memory requirement evaluation of applications is a major issue in the design of computer systems, and specifically in the case of embedded systems. We have shown for several application examples that this problem can often consist in maximizing a parametrized and multivariate polynomial defined over a parametrized convex domain. We proposed an original approach based on Bernstein expansion to compute accurate bounds for such polynomials, and even exact bounds in some cases. It has been implemented and is freely available.

We have also shown that static analysis of programs can provide high quality results for complicated and critical issues as soon as efficient mathematical tools are well used and adapted. Static analysis is superior to dynamic and experimental approaches, such as profiling or iterative compilation, since it can directly provide accurate and correct results. Moreover, these results, when parametrized, can cover all possible execution configurations from only one unique program analysis process.

In this paper, Bernstein expansion is used to analyze polynomials resulting from previous program analysis steps. However, Bernstein polynomials can also be used to perform polynomial interpolation. We are currently investigating the use of Bernstein interpolation to model data that is too difficult or too complex to be handled directly, enabling some new interesting program transformations.

REFERENCES

Barvinok, A. and Pommersheim, J. 1999. An algorithmic theory of lattice points in polyhedra. *New Perspectives in Algebraic Combinatorics 38*, 91–147.

Bastoul, C. 2004. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, USA, 7–16.

Bauer, C., Frink, A., and Kreckel, R. 2002. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *J. Symb. Comput. 33,* 1, 1–12.

Berchtold, J. and Bowyer, A. 2000. Robust arithmetic for multivariate bernstein-form polynomials. *Computer-aided Design 32*, 681–689.

Bernstein, S. 1952. *Collected Works*. Vol. 1. USSR Academy of Sciences.

Bernstein, S. 1954. *Collected Works*. Vol. 2. USSR Academy of Sciences.

Beyls, K. and D'Hollander, E. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture 51,* 4 (4), 223–250.

Blanchet, B. 1999. 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*. ACM Press, 20–34.

Blume, W. and Eigenmann, R. 1994. Symbolic range propagation. Tech. Rep. 1381, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev. Oct.

Blume, W. and Eigenmann, R. 1995. Symbolic range propagation. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 357–363.

Braberman, V., Garbervetsky, D., and Yovine, S. 2006. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology 5,* 5 (jun), 31–58. http://www.jot.fm/issues/issue_2006_06/article2.pdf.

Cherem, S. and Rugina, R. 2004. Region analysis and transformation for java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*. ACM Press, New York, NY, USA, 85–96.

Chin, W.-N., Nguyen, H. H., Qin, S., and Rinard, M. C. 2005. Memory usage verification for

OO programs. In *Static Analysis, 12th International Symposium, SAS 2005*, C. Hankin and I. Siveroni, Eds. Lecture Notes in Computer Science, vol. 3672. Springer, 70–86.

Clauss, P. and Loechner, V. 1998. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing 19,* 2, Kluwer Academic.

Clauss, P. and Tchoupaeva, I. 2004. A symbolic approach to bernstein expansion for program analysis and optimization. In *13th International Conference on Compiler Construction, CC 2004*, E. Duesterwald, Ed. LNCS, vol. 2985. Springer, 120–133.

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

De Loera, J. A., Hemmecke, R., Köppe, M., and Weismantel, R. 2006. Integer polynomial optimization in fixed dimension. *Math. Oper. Res. 31,* 1 (Feb.), 147–153.

Fahringer, T. 1998. Efficient symbolic analysis for parallelizing compilers and performance estimators. *J. Supercomput. 12,* 3, 227–252.

Farin, G. 1993. *Curves and Surfaces in Computer Aided Geometric Design*. Academic Press, San Diego.

Farouki, R. and Rajan, V. 1987. On the numerical condition of polynomials in bernstein form. *Computer Aided Geometric Design 4,* 3, 191–216.

Feautrier, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program. 20,* 1, 23–53.

Feautrier, P. 1996. *The Data Parallel Programming Model*. LNCS, vol. 1132. Springer-Verlag, Chapter Automatic Parallelization in the Polytope Model, 79–100.

Fernández, F. J. 2006. Obtaining memory bounds of the required memory to run a method in the memory scoped model with bernstein basis. M.S. thesis, Departamento de Computación. FCEyN. Universidad de Buenos Aires.

Garbervetsky, D., Nakhli, C., Yovine, S., and Zorgati, H. 2004. Program instrumentation and run-time analysis of scoped memory in java. In *RV 2004: International Workshop on Runtime Verification*. ENTCS, vol. 113. ETAPS, Elsevier, Barcelona, Spain, 105–121.

Garloff, J. 1999. Application of bernstein expansion to the solution of control problems. In *Proceedings of MISC'99 - Workshop on Applications of Interval Analysis to Systems and Control*, J. Vehi and M. A. Sainz, Eds. University of Girona, Girona (Spain), Springer Netherlands, 421–430.

Garloff, J. and Graf, B. 1999. *The Use of Symbolic Methods in Control System Analysis and Design*. Institution of Electrical Engineers (IEE), London, Chapter Solving Strict Polynomial Inequalities by Bernstein Expansion, 339–352.

Gheorghioiu, O. 2002. Statically determining memory consumption of real-time java threads. M.S. thesis, Massachusetts Institute of Technology.

GMP. The GNU MP bignum library. http://www.swox.com/gmp/.

Grun, P., Balasa, F., and Dutt, N. 1998. Memory size estimation for multimedia applications. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*. IEEE Computer Society, Washington, DC, USA, 145–149.

Haible, B. 2006. *CLN*: Class library for numbers. Available at http://www.ginac.de/CLN/ .

Hofmann, M. and Jost, S. 2003. Static prediction of heap usage for first-order functional programs. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 185–197.

Hosangadi, A., Fallah, F., and Kastner, R. 2006. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25,* 10 (Oct.), 2012–2022.

Hughes, J. and Pareto, L. 1999. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *ICFP '99*. ACM, 70–81.

Hughes, J., Pareto, L., and Sabry, A. 1996. Proving the correctness of reactive systems using sized types. In *POPL '96*. ACM, 410–423.

KJELDSBERG, P. G., CATTHOOR, F., AND AAS, E. J. 2004. Storage requirement estimation for optimized design of data intensive applications. *ACM Trans. Des. Autom. Electron. Syst. 9,* 2, 133–158.

LOECHNER, V. 1999. Polylib: A library for manipulating parameterized polyhedra. Tech. rep., ICPS, Université Louis Pasteur de Strasbourg, France. Mar.

MARTIN, R., SHOU, H., VOICULESCU, I., BOWYER, A., AND WANG, G. 2002. Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design 19*, 553–587.

MASLOV, V. AND PUGH, W. 1994. Simplifying polynomial constraints over integers to make dependence analysis more precise. In *CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing.*

MEISTER, B. 2004. Stating and manipulating periodicity in the polytope model. applications to program analysis and optimization. Ph.D. thesis, ICPS, Université Louis Pasteur de Strasbourg, France.

RABL, T. 2006. Volume calculation and estimation of parameterized integer polytopes. M.S. thesis, Universität Passau.

RAMANUJAM, J., HONG, J., KANDEMIR, M. T., AND NARAYAN, A. 2001. Reducing memory requirements of nested loops for embedded systems. In *Design Automation Conference.* 359–364.

SALCIANU, A. AND RINARD, M. 2001. Pointer and escape analysis for multithreaded programs. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming.* ACM Press, 12–23.

SEGHIR, R. AND LOECHNER, V. 2006. 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.*

STAHL, V. 1995. Interval methods for bounding the range of polynomials and solving systems of nonlinear equations. Ph.D. thesis, Johannes Kepler University Linz, Austria.

TURJAN, A., KIENHUIS, B., AND DEPRETTERE, E. 2004. Translating affine nested-loop programs to process networks. In *Proc. International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'04).* Washington D.C., USA, 220–229.

UNNIKRISHNAN, L., STOLLER, S., AND LIU, Y. 2003. Optimized live heap bound analysis. In *VMCAI 03.* LNCS, vol. 2575. 70–85.

VAN ENGELEN, R., GALLIVAN, K., AND WALSH, B. 2003. Tight timing estimation with the Newton-Gregory formulae. In *10th Workshop on Compilers for Parallel Computers, CPC 2003.*

VERBAUWHEDE, I. M., SCHEERS, C. J., AND RABAEY, J. M. 1994. Memory estimation for high level synthesis. In *DAC '94: Proceedings of the 31st annual conference on Design automation.* ACM Press, New York, NY, USA, 143–148.

VERDOOLAEGE, S. 2005. Incremental loop transformations and enumeration of parametric sets. Ph.D. thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium.

VERDOOLAEGE, S. 2006. *barvinok*, a library for counting the number of integer points in parametrized and non-parametrized polytopes.
Available at http://freshmeat.net/projects/barvinok.

VERDOOLAEGE, S., BEYLS, K., BRUYNOOGHE, M., AND CATTHOOR, F. 2005. Experiences with enumeration of integer projections of parametric polytopes. In *Proceedings of 14th International Conference on Compiler Construction, Edinburgh, Scotland*, R. Bodík, Ed. Lecture Notes in Computer Science, vol. 3443. Springer, Berlin / Heidelberg, 91–105.

VERDOOLAEGE, S., BRUYNOOGHE, M., JANSSENS, G., AND CATTHOOR, F. 2003. Multi-dimensional incremental loop fusion for data locality. In *IEEE 14th International Conference on Application-specific Systems, Architectures and Processors*, D. Martin, Ed. The Hague, The Netherlands, 17–27.

VERDOOLAEGE, S., NIKOLOV, H., AND STEFANOV, T. 2006. Improved derivation of process networks. In *4th Workshop on Optimization for DSP and Embedded Systems, ODES-4.*

VERDOOLAEGE, S., SEGHIR, R., BEYLS, K., LOECHNER, V., AND BRUYNOOGHE, M. 2007. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica.* accepted for publication.

Zettler, M. and Garloff, J. 1998. Robustness analysis of polynomials with polynomial parameter dependency using bernstein expansion. *IEEE Transactions on Automatic Control 43,* 3, 425–431.

Zhao, Y. and Malik, S. 2000. Exact memory size estimation for array computations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8,* 5 (Oct.), 517–521.

Zhu, H. 2006. Computation of memory requirements for multi-dimensional signal processing applications. Ph.D. thesis, University of Illinois at Chicago. Preliminary Doctoral Thesis.

Zhu, H., Luican, I. I., and Balasa, F. 2006. Memory size computation for multimedia processing applications. In *ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation*. ACM Press, New York, NY, USA, 802–807.

Received Month Year; revised Month Year; accepted Month Year