Analytical Computation of Ehrhart Polynomials and its Application in Compile-Time Generated Cache Hints

Rachid Seghir
ICPS, LSIIT (UMR CNRS 7005)
Université Louis Pasteur, Strasbourg
seghir@icps.u-strasbg.fr

Sven Verdoolaege
Dept. of Computer Science
K.U.Leuven
sven.verdoolaege@cs.kuleuven.ac.be

Kristof Beyls
Dept. of Electronics and Information Systems
Ghent University
kristof.beyls@elis.UGent.be

Vincent Loechner
ICPS, LSIIT (UMR CNRS 7005)
Université Louis Pasteur, Strasbourg
loechner@icps.u-strasbg.fr

ABSTRACT

In modern micro-architectures, computation speed is often reduced by cache misses. Cache analysis is therefore imperative to obtain effective optimization. We present an analytical technique based on reuse distances that focuses on efficiently determining the behavior of fully associative caches and extends to set-associative caches. In this technique, the number of cache misses is obtained by counting the number of integer points in a parameterized polytope.

It is well know that this parameterized count can be represented by an Ehrhart polynomial. Previously, interpolation was used to obtain these polynomials, but this technique has some disadvantages, most notably that under certain conditions it fails to produce a solution. Our main contribution is a novel method for calculating Ehrhart polynomials *analytically*. It extends an existing method, based on Barvinok's decomposition, for counting the number of points in a non-parameterized polytope. Our technique always produces a solution and is usually faster than interpolation.

1. INTRODUCTION

With the performance gap between processor and memory doubling about every two years, current processors are able to execute up to 1000 instructions in the time of a single memory fetch. Although caches are used to alleviate this bottleneck, many programs do not sufficiently exploit the cache hardware. Large speedups can therefore be obtained for these programs by optimizing their cache behavior.

Two different techniques are used to analyze a program's cache behavior: profiling and analytical modeling. The first technique is based on cache simulators or locality analyzers and although this technique allows accurate measurement of the cache behavior, it remains slow [36, 23] and cannot be directly applied for different program inputs. In contrast, analytical methods use memory access patterns to predict the cache behavior [17, 20, 31]. In this paper, we present an analytical technique that differs from the previous ones in the fact that it allows to efficiently determine the behavior of fully associative caches and extends to set-associative ones. This method exploits the concept of *reuse distance* to communicate the locality of memory accesses to the processor through *cache hints* and leads to an improved LRU policy. The reuse distance of a certain access in a loop to a memory element A is the number of distinct memory elements accessed between this access and the next access to A and may depend on the iteration of the loop. This number is obtained by counting the integer points in a union of finite sets described by linear equations and parameterized by the iterators. Such sets are called *parameterized polytopes*. The result of this counting is represented by an *Ehrhart polynomial* [15, 11], which is a multivariate polynomial in the parameters. The coefficients of such polynomials depend periodically on the parameter values. Parameterized counting has many applications in the program optimization community. In particular, it is very useful in cache analysis [10, 11, 4], but has also found applications in the context of process networks [30, 35] and real-time systems [24, 8].

Several techniques have been proposed for counting the integer points in finite sets, but most do not handle parameterized problems symbolically (e.g., [28, 14]), while some others have never been implemented (e.g., [29]). Arguably the most general and useful technique to date which handles arbitrary parameterized polytopes is that of Clauss and Loechner [11] implemented in `PolyLib` [25]. Their method computes the coefficients of an Ehrhart polynomial by solving a system of equations based on the function values for some initial parameter values, in effect interpolating the polynomial. Sometimes the number of available parameter values is insufficient and then the technique fails to produce a solution. In our application we have seen that this failure occurs for three out of seven benchmark programs.

In this paper, we present a new counting method that also targets arbitrary parameterized polytopes. Unlike the previous method, ours calculates Ehrhart polynomials analytically. It therefore *always* produces a solution and it is usually also *faster* than the interpolation method. Our technique extends an existing, polynomial time (for fixed dimension) algorithm for computing the number of integer points in non-parameterized polytopes which is based on Barvinok's decomposition [2].

```
S1  do j = 1, N
S2    do l = j, N
S3      do k = 1, j-1
S4        A(l,j) = A(l,j) - A(l,k) * A(j,k)
S5      enddo
S6    enddo
S7    A(j,j) = sqrt(A(j,j))
S8    do m = j+1, N
S9      A(m,j) = A(m,j) / A(j,j)
S10   enddo
S11 enddo
```

Figure 1: Source code of Cholesky factorization

The remainder of this paper is organized as follows. In Section 2 we show through an example our motivation behind this work. In Section 3 we first recall both some notions from the method of Clauss and Loechner and Barvinok's algorithm. We then show how these two can be combined in our own method. In Section 4 we explain how we apply our counting method to improve the LRU policy through the use of reuse distances and cache hints and we give some experimental results. Finally, implementation details and related works are given in Section 5.

2. MOTIVATING EXAMPLE

In this section we present an example of polytope enumeration to improve cache accesses.

The *forward reuse distance* is the number of unique memory locations accessed by a program between two accesses to the same data element. This extra information can be added as an extra register operand to the instruction performing the first data access. This is called a *cache hint* and it is useful to improve the LRU replacement policy: if the forward reuse distance is larger than the cache size, then the data will probably not be reused and should be evicted more quickly from the cache. The LRU+CH (LRU+cache hints) replacement policy has been proven to perform at least as well as the LRU policy [21]. More experiments are given in section 4.

Let us consider the Cholesky factorization. The source code is presented in Figure 1. Consider the iteration spaces P_1 enclosing statement S4 and P_2 enclosing S9:

$$\begin{aligned} P_1 &= \{\, (j,l,k) \in \mathbb{Z}^3 \mid 1 \le j \le l \le N \wedge 1 \le k \le j-1 \,\} \\ P_2 &= \{\, (j,m) \in \mathbb{Z}^2 \mid 1 \le j \le N \wedge j+1 \le m \le N \,\}. \end{aligned}$$

We will compute the forward reuse distance for the data element $A(m,j)$ accessed by statement S9.

A dependence analysis yields to the following result. Statement S9 accesses $A(m_0,j_0)$ at iteration (j_0,m_0) of P_2, which is reused for the first time by statement S4 through reference $A(l_1,k_1)$ at iteration (j_1,l_1,k_1) of P_1, with $j_1 = j_0+1$, $l_1 = m_0$, and $k_1 = j_0$. The set of iterations between these two accesses is $P_2' \cup P_1'$ with:

$$\begin{aligned} P_2' &= \big\{\, (j,m) \in P_2 \mid \big(j = j_0 \wedge m \ge m_0\big) \,\big\} \\ P_1' &= \big\{\, (j,l,k) \in P_1 \mid \big(j = j_1 \wedge l < l_1\big) \vee \\ &\qquad \big(j = j_1 \wedge l = l_1 \wedge k \le k_1\big) \,\big\}. \end{aligned}$$

The next step consists in computing all the data accessed by this set of iterations. In iteration space P_2 the elements $A(m,j)$ and $A(j,j)$ are accessed, while in P_1 the elements $A(l,j)$, $A(l,k)$, and $A(j,k)$ are accessed. We assume that the order of accesses in statement S9 is $A(m,j)$; $A(j,j)$, and the order of accesses in statement S4 is $A(l,j)$; $A(l,k)$; $A(j,k)$. So the set of data being accessed in array A is:

$$\begin{aligned} &\big\{\, (m,j) \mid (j,m) \in P_2' \setminus \{(j_0,m_0)\} \,\big\} \\ \cup &\big\{\, (j,j) \mid (j,m) \in P_2' \,\big\} \\ \cup &\big\{\, (l,j) \mid (j,l,k) \in P_1' \,\big\} \\ \cup &\big\{\, (l,k) \mid (j,l,k) \in P_1' \setminus \{(j_1,l_1,k_1)\} \,\big\} \\ \cup &\big\{\, (j,k) \mid (j,l,k) \in P_1' \setminus \{(j_1,l_1,k_1)\} \,\big\}. \end{aligned}$$

Finally, the forward reuse distance for element $A(m,j)$ accessed by statement S9 is the number of integer points contained in this union of polytopes. The union can easily be simplified as a union of disjoint polytopes and the sum of the corresponding Ehrhart polynomials is: $j_0(m_0 - j_0 - 1) + N$.

For each data element being loaded into cache, if this number is larger than the cache size, it will probably not be reused and should be evicted from cache quickly. As a result, there is less cache pollution and the data elements that have a chance to be reused probably will be.

The last step of our method, computing the Ehrhart polynomials, sometimes failed with the interpolation based implementation. This is due to a lack of sufficient parameter values needed to complete the system of equations from which the Ehrhart polynomial is computed. This is why we developed a novel method for calculating Ehrhart polynomials, presented in the following section.

3. COUNTING PARAMETERIZED SETS

In this section, we consider the problem of counting the number of integer points in a parameterized polytope, which is a set of points bounded by linear equations:

$$P_{\mathbf{p}} = \{\, \mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \ge B\mathbf{p} + \mathbf{c} \,\}, \tag{1}$$

where A and B are integer matrices, $\mathbf{c}$ is an integer vector and $\mathbf{p}$ is a parameter vector, $\mathbf{p} = (p_1, p_2, \ldots, p_n) \in \mathbb{Z}^n$. Without loss of generality (see Section 5), we assume that P is full-dimensional, i.e., that the dimension of P is d as well. Results cited from other sources have been adapted accordingly.

It is well known [15] that the above count can be represented by an Ehrhart polynomial $\mathcal{E}(P;\mathbf{p})$, which is a pseudo-polynomial of degree at most d. The coefficients of a pseudo-polynomial depend periodically on the parameters and are called periodic numbers.

DEFINITION 1. *An n-periodic number $U(\mathbf{p})$ is a function $\mathbb{Z}^n \mapsto \mathbb{Z}$, such that $U(\mathbf{p}) = U(\mathbf{p}')$ whenever $p_i \equiv p_i' \mod q_i$, for $1 \le i \le n$ and q_i the period in dimension i. The lcm (least common multiple) of all q_i is called the period of $U(\mathbf{p})$.*

These periodic numbers can be represented by a lookup-table $U_{\mathbf{p}}$ such that $U(\mathbf{p}) = U_{\mathbf{p}}[p_1 \bmod q_1, \ldots, p_n \bmod q_n]$.

In the remainder of this section, we first recall how Ehrhart polynomials were calculated previously and the problems this entails. Then we show how we solved these problems by taking an algorithm for counting the number of points in non-parameterized polytopes based on Barvinok's decomposition and extending it to compute Ehrhart polynomials.

3.1 Validity Domains and Interpolation

A parameterized polytope (1) can also be represented by an *explicit* notation:

$$P_{\mathbf{p}} = \left\{ \mathbf{x} \in \mathbb{Q}^d \mid \mathbf{x} = \boldsymbol{\lambda} V(\mathbf{p}), 0 \leq \lambda_j, \sum_j \lambda_j = 1 \right\}, \quad (2)$$

with the columns of $V(\mathbf{p})$ the extremal points of the polytope, called the *vertices* $\mathcal{V}(P_{\mathbf{p}})$. Each vertex $\mathbf{v}_j(\mathbf{p})$ is an affine combination of the parameters with *rational* coefficients. As Loechner and Wilde [26] showed, however, some of the vertices may only be valid for a subset of the parameter values. Therefore, the parameter space has to be partitioned into *validity domains* $\{ D_k \} = \mathcal{W}(P_{\mathbf{p}})$, each with a subset $\mathcal{V}_{D_k}(P_{\mathbf{p}}) \subset \mathcal{V}(P_{\mathbf{p}})$ of the total number of parameterized vertices.

Clauss and Loechner [11] showed that the period in dimension i of the periodic numbers that appear in the Ehrhart polynomial of a parameterized polytope (in a given validity domain) is a divisor of the lcm of the denominators of the coefficients of p_i in the affine expressions that define the vertices $\mathbf{v}_j(\mathbf{p})$ (that are valid in the validity domain). Based on this knowledge, they calculate the number of points in a set of instances of $P_{\mathbf{p}}$ for fixed values of $\mathbf{p}$ in a given validity domain, called *initial countings*, and then calculate the Ehrhart polynomial for this validity domain through interpolation.

The main problem with this approach is that the number of initial countings required for interpolating a d-dimensional Ehrhart polynomial with periods q_i is $\prod_{i=1}^{n}(d+1)q_i$. If a suitable set of parameter values cannot be found inside the validity domain, then this approach fails to produce a solution.

3.2 Barvinok's Algorithm

The basic idea behind Barvinok's algorithm [1, 9, 14], is to consider the *generating function* of the integer points in a polytope P. This generating function is a formal power series with a term for each integer point in P, i.e.,

$$f(P; \mathbf{x}) = \sum_{\boldsymbol{\alpha} \in P \cap \mathbb{Z}^d} \mathbf{x}^{\boldsymbol{\alpha}},$$

with $\mathbf{x}^{\boldsymbol{\alpha}} = x_1^{\alpha_1} x_2^{\alpha_2} \cdots x_d^{\alpha_d}$. Evaluating this function at $\mathbf{x} = \mathbf{1}$ yields the number of terms, which equals the desired number of points. The generating function is obviously not constructed by enumerating all the integer points in P, but rather as a signed sum of short rational functions that can be derived from the description of P.

Example 1. *Consider the polytope T shown in Figure 2: $T = \{\mathbf{x} | x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_1 + x_2 \leq 2\}$. Its generating function is $f(T; \mathbf{x}) = 1 + x_1 + x_1^2 + x_2 + x_1 x_2 + x_2^2$. Barvinok's algorithm, however, will produce this function in the following form:*

$$\frac{x_2^2}{(1 - x_2^{-1})(1 - x_1 x_2^{-1})} + \frac{x_1^2}{(1 - x_1^{-1})(1 - x_1^{-1} x_2)} + \frac{1}{(1 - x_1)(1 - x_2)}.$$

Computing the residue at $\mathbf{1}$, we can confirm that the number of integer points in T is indeed $f(T; \mathbf{1}) = 6$.

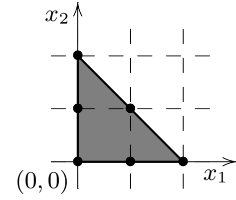


Figure 2: Barvinok example

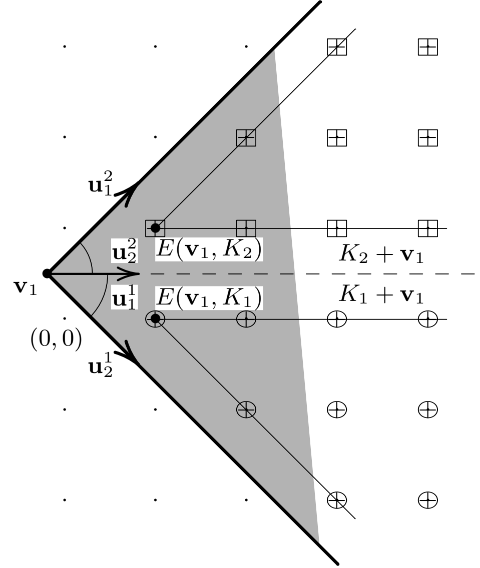


Figure 3: Barvinok's decomposition of cone$(P, \mathbf{v}_1)$.

To construct the generating function as a signed sum of short rational functions, we consider the vertices $\mathbf{v}_i$ of P and the constraints that it saturates, i.e., the constraints $\langle \mathbf{a}, \mathbf{x} \rangle \geq b$ with $\langle \mathbf{a}, \mathbf{v}_i \rangle = b$, where $\langle \cdot, \cdot \rangle$ is the standard scalar product. The region in $\mathbb{Q}^d$ bounded by these constraints for a particular $\mathbf{v}_i$ is called the *supporting cone* cone$(P, \mathbf{v}_i)$. E.g., the supporting cone of vertex $\mathbf{v}_1$ of the polytope (shaded area) in figure 3 is shown in thick lines. It can be shown [1] that the generating function of P is equal to the sum of the generating functions of its supporting cones. To construct the generating function of a supporting cone, we use Barvinok's decomposition into *unimodular cones*.

Definition 2. *A* **cone** *with* **generators** $\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k \in \mathbb{Z}^d$ *is a set of the form $\{ \sum_i \lambda_i \mathbf{u}_i \mid \lambda_i \geq 0 \}$. It is called* ***unimodular*** *if its generators form a basis of $\mathbb{Z}^d$.*

Note that using this definition, $\text{cone}(P, \mathbf{v}_i)$ is not a cone itself, but the sum of $\mathbf{v}_i$ and some cone K. Barvinok proposed to decompose this cone into a signed "sum" of unimodular cones $\{(\epsilon_i, K_i)\} = \mathcal{B}(K)$, with $\epsilon_i \in \{-1, 1\}$, the sign corresponding to unimodular cone K_i. Here, "sum" can be interpreted to mean that the generating function of K is the signed sum of the generating functions of the unimodular cones. It can be shown [1] that a simple explicit formula exists for the generating function of a unimodular cone: $f(K_i; \mathbf{x}) = \prod_{j=1}^{k}(1 - \mathbf{x}^{\mathbf{u}_j^i})^{-1}$, with $\mathbf{u}_j^i$ the generators of K_i. A key feature of this decomposition is that it takes polynomial time (for fixed dimensions).

To obtain the final generating function, the generating functions corresponding to the unimodular cones K_i need to be translated to the vertex $\mathbf{v}$. If $\mathbf{v}$ is integer, then we simply need to add $\mathbf{v}$ to all of the exponents in the generating function, which corresponds to a multiplication by $\mathbf{x}^{\mathbf{v}}$. If $\mathbf{v}$ is not integer, however, then we need to find another point $\mathbf{v}' = E(\mathbf{v}, K_i)$ such that $\mathbf{x}^{\mathbf{v}'} f(K_i; \mathbf{x})$ generates $K_i + \mathbf{v}$. Since K_i is unimodular, this point exists and is uniquely defined as the smallest integer linear combination of the generators of K_i that lies inside $K_i + \mathbf{v}$ [14]. I.e.,

$$E(\mathbf{v}, K_i) = \sum_j \lceil \lambda_j \rceil \mathbf{u}_j^i, \tag{3}$$

where $\boldsymbol{\lambda}$ is the rational solution to $\mathbf{v} = \sum_j \lambda_j \mathbf{u}_j^i$ and $\lceil . \rceil$ is the upper integer part. Note that if $\mathbf{v}$ is integer, then $E(\mathbf{v}, K_i) = \mathbf{v}$. The final generating function is then

$$f(P; \mathbf{x}) = \sum_{\mathbf{v} \in \mathcal{V}(P)} \sum_{i=1}^{|\mathcal{B}(K_{\mathbf{v}})|} \epsilon_i \frac{\mathbf{x}^{E(\mathbf{v}, K_i)}}{\prod_{j=1}^{d}(1 - \mathbf{x}^{\mathbf{u}_j^i})}. \tag{4}$$

Example 2. *Figure 3 shows a polytope P (shaded area) and its supporting cone (thick lines) at vertex $\mathbf{v}_1$. A possible signed unimodular decomposition for* $\text{cone}(P, \mathbf{v}_1) - \mathbf{v}_1$ *is the pair* $\{(1, K_1), (1, K_2)\}$.[1] *Let $\mathbf{v}_1' = E(\mathbf{v}_1, K_1)$ and $\mathbf{v}_1'' = E(\mathbf{v}_1, K_2)$. Since both signs are positive, we have*

$$f(\text{cone}(P, \mathbf{v}_1); \mathbf{x}) = f(\mathbf{v}_1' + K_1; \mathbf{x}) + f(\mathbf{v}_1'' + K_2; \mathbf{x}).$$

The integer points in $\mathbf{v}_1' + K_1$ are indicated by $\boxplus$, whereas the integer points in $\mathbf{v}_1' + K_1$ are indicated by $\oplus$.

Note that each term in (4) has a pole at $\mathbf{x} = \mathbf{1}$. We can still evaluate this function at $\mathbf{x} = \mathbf{1}$ by computing the residue. De Loera [14] shows that, through a suitable variable substitution, each term in (4) can be written as

$$\epsilon_i' \frac{N(s)}{D'(s)} = \epsilon_i' \frac{(s+1)^{\langle \boldsymbol{\mu}, E(\mathbf{v}, K_i) \rangle + c}}{s^d D(s)}, \tag{5}$$

with $\mathbf{x} = (s+1)^{\boldsymbol{\mu}}$, $D(s)$ a polynomial with integer coefficients, independent of $\mathbf{v}$, $\boldsymbol{\mu}$ some integer vector and $\mathbf{c}$ some integer constant. Evaluating (4) at $\mathbf{x} = \mathbf{1}$ is equivalent to summing the terms (5) evaluated at $s = 0$. This in turn can be accomplished by computing the coefficient of s^d in the Taylor expansion of $N(s)/D(s)$. Note that this only requires the first $(d+1)$ coefficients of $N(s)$. The (signed) sum of the coefficients of s^d in each of the terms then yields the desired number of points in the polytope.

[1] Note that this is not the decomposition that our implementation would produce, since, like De Loera [14], we perform the decomposition on the dual cone.

Algorithm 1 Parameterized Barvinok

1. For each vertex $\mathbf{v}_i(\mathbf{p}) \in \mathcal{V}(P)$
   (a) Determine supporting cone $\text{cone}(P, \mathbf{v}_i(\mathbf{p}))$
   (b) Let $K = \text{cone}(P, \mathbf{v}_i(\mathbf{p})) - \mathbf{v}_i(\mathbf{p})$
   (c) Let $\{(\epsilon_j, K_j)\} = \mathcal{B}(K)$
   (d) For each K_j
      i. Determine $f(K_j; \mathbf{x})$
   (e) $f(\text{cone}(P, \mathbf{v}_i(\mathbf{p})); \mathbf{x}) = \sum_j \epsilon_j \mathbf{x}^{E(\mathbf{v}_i(\mathbf{p}), K_j)} f(K_j; \mathbf{x})$
2. For each validity domain D_k of P
   (a) $f_{D_k}(P; \mathbf{x}) = \sum_{\mathbf{v}_i \in \mathcal{V}_{D_k}(P)} f(\text{cone}(P, \mathbf{v}_i(\mathbf{p})); \mathbf{x})$
   (b) evaluate $f_{D_k}(P; \mathbf{1})$

3.3 Computing Ehrhart Polynomials

Algorithm 1 shows our extension of Barvinok's method to parameterized polytopes. The main idea behind this generalization is to consider Loechner and Wilde's decomposition of the parameter space and to apply Barvinok's algorithm to the fixed set of (parameterized) vertices that belong to each validity domain. Thus, one parameterized generating function is to be computed for each of these validity domains.

Similarly to the non-parameterized case, the generating function for the parameterized polytope $P_{\mathbf{p}}$ on validity domain D is the parameterized version of equation (4):

$$f_D(P_{\mathbf{p}}; \mathbf{x}) = \sum_{\mathbf{v}(\mathbf{p}) \in \mathcal{V}_D(P_{\mathbf{p}})} \sum_{i=1}^{|\mathcal{B}(K_{\mathbf{v}})|} \epsilon_i \frac{\mathbf{x}^{E(\mathbf{v}(\mathbf{p}), K_i)}}{\prod_{j=1}^{d}(1 - \mathbf{x}^{\mathbf{u}_j^i})}, \tag{6}$$

with $\epsilon_i \in \{-1, 1\}$ and $\mathbf{v}(\mathbf{p})$ a parameterized vertex of the polytope $P_{\mathbf{p}}$. Each coordinate of $\mathbf{v}(\mathbf{p})$ is an affine function of the parameters. K_i is the ith unimodular cone in the signed unimodular decomposition of cone $K_{\mathbf{v}}$, the translation to the origin of the supporting cone at $\mathbf{v}(\mathbf{p})$. The supporting cone is again defined by the constraints that $\mathbf{v}(\mathbf{p})$ saturates, i.e., the constraints $\langle \mathbf{a}, \mathbf{x} \rangle \geq \langle \mathbf{b}, \mathbf{p} \rangle + c$ such that $\langle \mathbf{a}, \mathbf{v}(\mathbf{p}) \rangle = \langle \mathbf{b}, \mathbf{p} \rangle + c$. The correctness of (6) follows from the fact that the generators of K are independent of the parameters, which means that Barvinok's decomposition can be applied without change.

The exponent in the numerators of (6), which corresponds to the uniquely defined point inside the translated unimodular cone, is given by the parameterized version of (3):

$$E(\mathbf{v}(\mathbf{p}), K_i) = \sum_{j=1}^{d} \lceil \lambda_j(\mathbf{p}) \rceil \mathbf{u}_j^i, \tag{7}$$

where the $\lambda_j(\mathbf{p})$s are rational affine functions of the parameters that solve $\mathbf{v}(\mathbf{p}) = \sum_{j=1}^{d} \lambda_j(\mathbf{p}) \mathbf{u}_j^i$.

Let m be an integer constant such that $m\lambda_j(\mathbf{p})$ is an integer

affine function, then [18]

$$\lceil \lambda_j(\mathbf{p}) \rceil = \left\lceil \frac{m\lambda_j(\mathbf{p})}{m} \right\rceil = \lambda_j(\mathbf{p}) + \frac{(-m\lambda_j(\mathbf{p})) \bmod m}{m}. \quad (8)$$

The second term on the right is a periodic number, say $U'_j(\mathbf{p})$, with period at most m [27]. As explained near the top of Section 3, this periodic number can be represented by a lookup-table. To construct this table, we simply evaluate the modulo expression in (8) for a set of fixed parameter values. Note that, unlike was the case with interpolation, we need not restrict ourselves to values from the validity domain since the expression in (8) is valid for all values of $\mathbf{p}$. Substituting the value of $\lceil \lambda_j(\mathbf{p}) \rceil$ in (7) we get

$$E(\mathbf{v}(\mathbf{p}), K_i) = \sum_{j=1}^{d} \lambda_j(\mathbf{p}) u_j^i + \sum_{j=1}^{d} U'_j(\mathbf{p}) u_j^i = \mathbf{v}(\mathbf{p}) + \mathbf{U}(\mathbf{p}), \quad (9)$$

with $\mathbf{U}(\mathbf{p})$ a vector of periodic numbers.

As in the non-parameterized case, we can obtain the value of the parameterized generating function (6) at $\mathbf{x} = \mathbf{1}$ by computing the residue. Again, the variable substitution proposed by De Loera [14] is independent of the numerator and hence of the parameters. Substituting (9) in (5), we obtain

$$N_{\mathbf{p}}(s) = (s+1)^{\langle \boldsymbol{\mu}, \mathbf{v}(\mathbf{p}) + \mathbf{U}(\mathbf{p}) \rangle + c} = (s+1)^{\Lambda(\mathbf{p})},$$

with $\Lambda(\mathbf{p})$ an affine function of the parameters with a constant part that may be a periodic number. The coefficients of $N_{\mathbf{p}}(s)$ up to that of s^d (i.e., those required to compute the coefficient of s^d in $N_{\mathbf{p}}(s)/D(s)$) are

$$n_i(\mathbf{p}) = \binom{\Lambda(\mathbf{p})}{i} = \frac{\prod_{j=0}^{i-1}(\Lambda(\mathbf{p}) - j)}{i!} \quad \text{for } 0 \le i \le d.$$

Each coefficient $n_i(\mathbf{p})$ in the above formula is given by a product of at most d affine functions of the parameters with constant parts that may be periodic numbers. This implies that each of these coefficients is a multivariate polynomial of the parameters in which the coefficients may be periodic numbers and for which the sum of powers in each multivariate monomial is at most d. Since the coefficient of s^d in $N_{\mathbf{p}}(s)/D(s)$ is a linear combination of these $n_i(\mathbf{p})$, it conforms to the same property and so does the (signed) sum of all these terms. That is, the residue of (6), which is equal to the (parameterized) number of points in $P_{\mathbf{p}}$, is an Ehrhart polynomial, as expected. I.e.,

$$\mathcal{E}_D(P; \mathbf{p}) = f_D(P_{\mathbf{p}}; \mathbf{1}) = \sum_{0 \le i_1 + i_2 + \cdots + i_n \le d} U_{\mathbf{i}}(\mathbf{p}) \mathbf{p}^{\mathbf{i}},$$

with the $U_{\mathbf{i}}(\mathbf{p})$s periodic numbers and d the dimension of $P_{\mathbf{p}}$. Note that this Ehrhart polynomial is calculated *analytically* and that it can be computed for *any* validity domain, no matter its size or shape.

4. CACHE HINT SELECTION BASED ON REUSE DISTANCE EQUATIONS

In this section, a new compiler optimization enabled by the analytical calculation of Ehrhart polynomials is described. The aim of the optimization is to improve the replacement decisions in the data cache, by communicating the locality of memory accesses to the processor through cache hints.

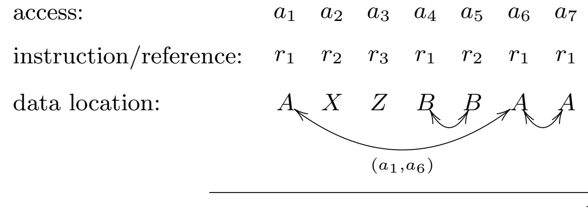


Figure 4: The top row indicates 7 sequential memory accesses, which are generated by the references in the second row. The bottom row shows the corresponding memory locations A, B, X or Z. The accesses to X and Z are not part of a reuse pair, since they are accessed only once in the stream. $\mathsf{ADS}(a_1, a_6) = \{B, X, Z\}$, and $\mathsf{RD}(a_1, a_6) = |\mathsf{ADS}(a_1, a_6)| = 3$. $\mathsf{RD}(a_6, a_7) = 0$. $\mathsf{FRD}(a_1) = 3$. $\mathsf{FRD}(a_6) = 0$.

4.1 Reuse Distance and Cache Hints

DEFINITION 3. *A* **memory reference** *corresponds to a read or write instruction, while a particular execution of that read or write at runtime is a* **memory access** *[17]. A* **reuse pair** *(a_1, a_2) is a pair of memory accesses in a memory access stream, which touch the same memory location, without intermediate accesses to that location. The* **accessed data set (ADS)** *of a reuse pair (a_1, a_2) is the set of unique memory locations accessed between a_1 and a_2 and is denoted by $\mathsf{ADS}(a_1, a_2)$. The* **reuse distance** *of a reuse pair (a_1, a_2) is the number of unique memory locations accessed between accesses a_1 and a_2. It is denoted by $\mathsf{RD}(a_1, a_2)$, and equals $|\mathsf{ADS}(a_1, a_2)|$. The* **forward reuse distance** *of a memory access a_1 is the reuse distance of the pair (a_1, a_2). If there is no such reuse pair, its forward reuse distance is ∞.*

LEMMA 1. *In a fully associative LRU cache with CS lines, the memory line referenced by an access a will stay in the cache until the next use of that memory line if and only if $\mathsf{FRD}(a) < \mathsf{CS}$.*

DEFINITION 4. *A* **cache hint** *is an extra register operand to a memory instruction, that contains the forward reuse distance of that memory access.*

EXAMPLE 3. *In the instruction* `ld r7=[r4], frd=r9`*, register* `r9` *is the cache hint. The contents of the register should be the forward reuse distance of that particular execution of the load-instruction.*

The cache replacement policy uses the cache hints to improve over the LRU replacement policy. When the FRD is larger than the cache size, the data will probably not be reused and should be evicted from the cache, so that additional cache space is available for other data. The LRU+CH replacement policy is presented in algorithm 2. It is based on the LRU policy, but gives accesses a for which the $\mathsf{FRD}(a) \ge \mathsf{CS}$ a low priority. In [21], Jain et al. proved that this replacement policy is guaranteed to perform at least as good as LRU.

Algorithm 2 LRU+CH Replacement Policy

1. if l is not present in the cache, replace the cache line at the bottom of the stack with line l.
2. if $\mathsf{FRD}(a) < \mathsf{CS}$, move l to the top of the stack, else if $\mathsf{FRD}(a) \geq \mathsf{CS}$, move l to the bottom of the stack.

4.2 Reuse Distance Equations

The reuse distances of the individual references in the program are calculated in 3 steps:

1. The reuse pairs in the memory access stream are calculated. For every two *references* (r, s), a set of polytopes $\mathsf{reuse}\,(r \rightarrow s)$ is generated, which represent all reuse pairs for which the first access is generated by an execution of reference r, and the second access is generated by s.
2. For each set of reuse pairs, a set of polytopes is constructed which describes the accessed data set (ADS) of the reuse pairs in the set.
3. The number of different memory locations in the ADS is counted, which equals the reuse distance of the reuse pair.

In the reuse distance equations, the following notations are used.

Definition 5. *The* **set of all the references** *in a program is denoted by* $\boldsymbol{\mathcal{R}}$. *The* **set of variables** *in a program is denoted by* $\boldsymbol{\mathcal{V}}$. *The* **iteration space** *of the statement in which a reference r occurs is denoted by* $\mathsf{IS}(\boldsymbol{r})$. *The* **memory location** *which is accessed by r at iteration point i is denoted by* $\boldsymbol{r@i}$. *The fact that iteration point i of reference r is* **executed before** *iteration point j of reference s is expressed as* $\boldsymbol{i_r \lessdot j_s}$. *The set of the program parameters is denoted by* $\boldsymbol{\mathcal{P}}$.

4.2.1 Reuse pair formulas

Every memory access is uniquely defined by the reference r which generates the access and the iteration point I_r at which the access occurs.

All reuse pairs (x, y) for which the first access x originates from reference r and the second access y originates from reference s, are combined into the set of reuse pairs denoted by $\mathsf{reuse}\,(r \rightarrow s)$, which contains the iteration points I_r and J_s that generate a reuse. These iteration points are described by the following simultaneous equations:

$$\forall r, s \in \mathcal{R} : \mathsf{reuse}\,(r \rightarrow s) = \{\, (I_r, J_s) \in \mathbb{Z}^n : \text{subject to conditions (10a)–(10d)} \,\}$$

$$I_r \in \mathsf{IS}(r) \wedge J_s \in \mathsf{IS}(s) \qquad (\textit{iteration space}) \qquad (10a)$$

$$I_r \lessdot J_s \qquad (\textit{execution ordering}) \qquad (10b)$$

$$r@I_r = s@J_s \qquad (\textit{same location}) \qquad (10c)$$

$$\forall t \in \mathcal{R} : \neg\, (\exists K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s \wedge t@K_t = r@I_r) \qquad (\textit{no intervening access}) \qquad (10d)$$

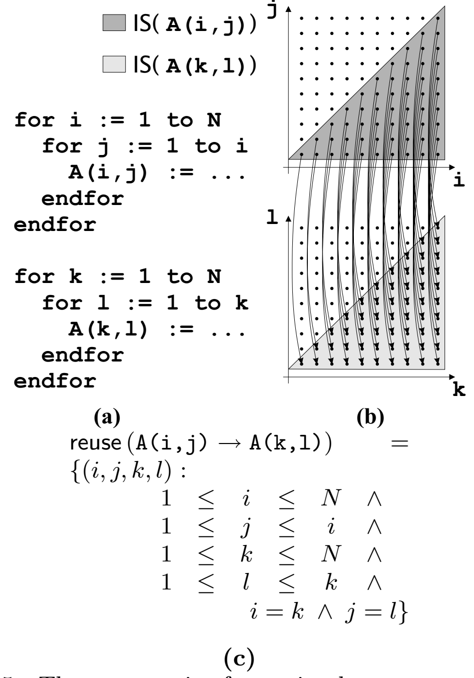


Figure 5: The reuse pairs for a simple program. In (a), the example program is shown. In (b), the reuse pairs are shown as arrows between the iteration points of the two different references. In (c), the reuse pairs are described by parameterized integer polytopes.

The above formula gives the constraints which must be satisfied before a reuse occurs between $r@I_r$ and $s@J_s$. Equation (10a) expresses that I_r and J_s are part of the iteration space of respectively r and s. (10b) demands that I_r must be executed before J_s; (10c) encodes that the same memory location must be accessed and (10d) ensures that no intervening memory access touches the same memory location.

An example of the above equations for a simple program is shown in Figure 5.

4.2.2 Accessed data set of a reuse pair

The function map_r^A maps an iteration space to the elements of array A accessed by r, while $\mathsf{iters}_t(I_r, J_s)$ is the set of iterations of reference t executed between iteration I_r and iteration J_s:

$$\mathsf{map}_r^A = \{I \rightarrow r@I : I \in \mathsf{IS}(r)\} \qquad (11)$$

$$\mathsf{iters}_t(I_r, J_s) = \{K_t \in \mathsf{IS}(t) : I_r \lessdot K_t \lessdot J_s\} \qquad (12)$$

Now $\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s))$, the elements of A that are in the ADS of the reuse pairs in $\mathsf{reuse}\,(r \rightarrow s)$, is expressed as follows:

$$\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s)) = \bigcup_{t \in \mathcal{R}} \mathsf{map}_t^A(\mathsf{iters}_t(\mathsf{reuse}\,(r \rightarrow s))), \qquad (13)$$

Equation (13) expresses that the ADS of a reuse pair can

be found by first calculating the iterations between use and reuse. Then, the ADS is simply all the data locations which are touched by the accesses in the iterations between use and reuse. The calculation of the ADS for a reuse pair of the program in figure 5(a) is shown in figure 6.

4.2.3 *Reuse distance of a reuse pair*

In order to find the reuse distance of a reuse pair, the number of different memory locations in its ADS needs to be counted:

$$\mathsf{RD}(\mathsf{reuse}\,(r \rightarrow s)) = \sum_{A \in \mathcal{V}} \mathcal{E}\left(\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s))\,; I_r, J_s, \mathcal{P}\right) \tag{14}$$

$\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s))$ is a set of parameterized integer polytopes. Besides calculating the reuse distance of a reuse pair, it is also possible to compute the forward reuse distances of a memory reference r, denoted by $\mathsf{FRD}(r)$:

$$\mathsf{FRD}(r) = \sum_{s \in \mathcal{R}, A \in \mathcal{V}} \mathcal{E}\left(\mathsf{ADS}^A\,(\mathsf{reuse}\,(r \rightarrow s))\,; I_r, \mathcal{P}\right) \tag{15}$$

Examples of the above equations are given in figure 7.

4.3 Experiments

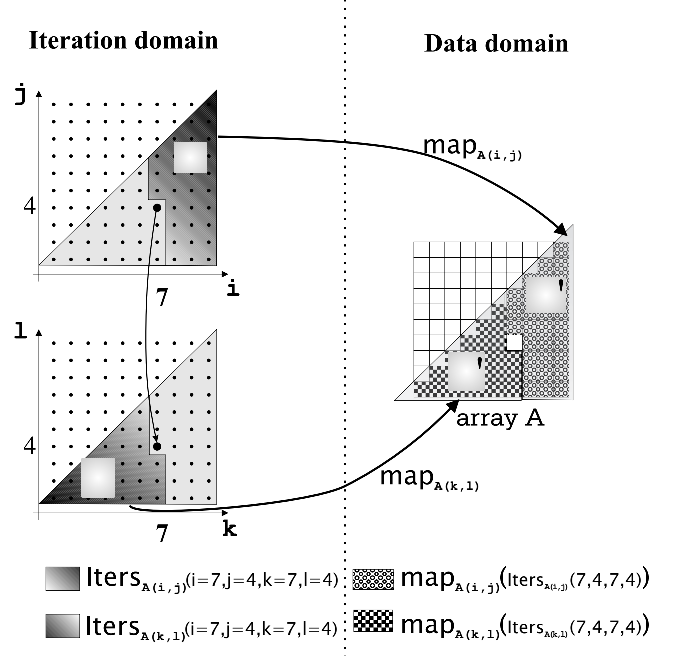
The reuse distance equations have been implemented in the FPT[38] compiler. The equations (10a)-(13) may consist of integer linear inequalities, the logical connectives $\neg, \vee, \wedge$ and the quantifier $\exists$. Such formulas are called Presburger formulas. The Omega library[22] is used to simplify them into a union of disjoint polytopes, which are counted using the method described in Section 3.

After the reuse distances have been calculated, FPT writes out an instrumented version of the source code, which produces a function call to a cache simulator for each memory access. The function call takes as arguments the address of the accessed variable and the calculated forward reuse distance. For example, the reuse distance $j(m - j - 1) + N$ calculated in section 2 for reference `A(m,j)` would show up in the instrumented code as:

```
...
do m = j+1, N
  CALL ACCESS(A(m,j), j*(m-j-1)+N)
  A(m,j) = ...
enddo
...
```

The calculated reuse distances were compared with reuse distances measured by profiling[5] and were found to be identical in all cases, indicating the exactness of our method.

For a number of programs from the NAS and SPEC benchmarks and Livermore loop kernels, this instrumentation has been performed. The number of polytopes that were enumerated are listed in table 1. It shows that for 3 programs, the previous method[11] fails to find the solution for all polytopes. Furthermore, the proposed method computes the number of points in a polytope about 3.7 times faster than the previous method.

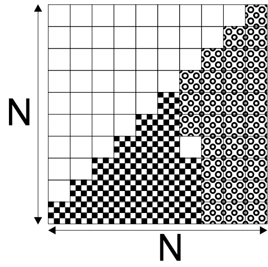


(a)

$$
\begin{aligned}
&\mathsf{ADS}^{\mathsf{A}}\,(\mathsf{reuse}\,(\mathtt{A(i,j)} \rightarrow \mathtt{A(k,l)})) = \\
&\big\{(x,y) : \big(1 \leq i \leq N \wedge 1 \leq j \leq i \,\wedge && \textit{IS}(\mathtt{A(i,j)}) \\
&\qquad 1 \leq k \leq N \wedge 1 \leq l \leq k \,\wedge && \textit{IS}(\mathtt{A(k,l)}) \\
&\qquad i = k \wedge j = l\big) \,\wedge && \textit{same location} \\
&\qquad \big(x < k \vee x = k \wedge y < l \,\vee && \textit{data accessed be-} \\
&\qquad x > i \vee x = i \wedge y > j\big)\big\} && \textit{tween use and reuse}
\end{aligned}
$$

(b)

Figure 6: Graphical representation of the calculation of $\mathsf{ADS}(\mathsf{reuse}\,(\mathtt{A(i,j)} \rightarrow \mathtt{A(k,l)}))$, for the program in figure 5(a). A single reuse pair is shown (from reference `A(i,j)` at iteration point $(i{=}7, j{=}4)$ to the access made by reference `A(k,l)` at iteration point $(k{=}7, l{=}4)$). On the left hand side, $\mathsf{iters}_{\mathtt{A(i,j)}}(i{=}7, j{=}4, k{=}7, l{=}4)$ and $\mathsf{iters}_{\mathtt{A(k,l)}}(i{=}7, j{=}4, k{=}7, l{=}4)$ are indicated in the iteration spaces of the references as areas α and β. After applying the mapping functions $\mathsf{map}^{\mathsf{A}}_{\mathtt{A(i,j)}}$ and $\mathsf{map}^{\mathsf{A}}_{\mathtt{A(k,l)}}$, the parts of array A that are accessed by the iterations between $(i{=}7, j{=}4)$ and $(k{=}7, l{=}4)$, are shown as α' and β'.



(a)

- $\mathsf{RD}(\mathsf{reuse}\,(\mathtt{A(i,j)} \to \mathtt{A(k,l)})) =$
  $\mathcal{E}\left(\mathsf{ADS}^{\mathtt{A}}\,(\mathsf{reuse}\,(\mathtt{A(i,j)} \to \mathtt{A(k,l)}))\,; \mathtt{i},\mathtt{j},\mathtt{k},\mathtt{l},\mathtt{N}\right) =$
  $\dfrac{N^2+N}{2} - 1$
- $\mathsf{FRD}(\mathtt{A(i,j)}) = \dfrac{N^2+N}{2} - 1$

(b)

Figure 7: In figure **(a)**, the accessed data elements of array A between use and reuse for the reuse in figure 6 is shown graphically. The amount of data in this set is $\frac{N^2+N}{2} - 1$, which equals the reuse distance of that reuse. In figure **(b)**, reuse distance and the forward reuse distance are described in function of matrix size N, using the equations (14)- (15).

The resulting program was linked with a cache simulator that simulates both the LRU and LRU+CH replacement policies. Table 2 shows the cache miss reduction for a 16KB 4-way set associative cache, resulting from the LRU+CH replacement policy, with the cache hints based on the forward reuse distance equations. After cache hint insertion, about 10% less cache misses occur.

5. IMPLEMENTATION DETAILS AND RELATED WORK

In Section 3, we assumed that P is full-dimensional. If P is of dimension $d - l$, with $l \geq 1$, then its description contains an equality $\langle \mathbf{a}, \mathbf{x} \rangle = \langle \mathbf{b}, \mathbf{p} \rangle + c$. Let $\mathbf{a}' = \mathbf{a}/g$, with g the gcd (greatest common divisor) of the elements in $\mathbf{a}$.

program	miss rate LRU	miss rate LRU+CH	miss rate reduction
vpenta	31.56%	25.57%	18.99%
mxm	3.20%	3.20%	0.00%
liv18	68.46%	61.91%	9.57%
cholesky	19.81%	17.94%	9.43%
jacobi	14.32%	14.32%	0.00%
gauss-jordan	11.90%	7.81%	34.37%
tomcatv	9.22%	9.22%	0.00%
average	22.64%	20.00%	**10.34%**

Table 2: The cache miss rates for a 4-way set associative 16KB cache with 32 bytes per line. The first column indicates the program name; the second column the miss rate with LRU replacement; the third column the miss rate of the LRU+CH replacement algorithm; and the fourth column shows the reduction in number of misses due to cache hints.

$\mathbf{a}'^T$ can be extended to a unimodular matrix U [6].[2] Let $P' = UP$. Since U and U^{-1} are unimodular, $\mathcal{E}(P') = \mathcal{E}(P)$. Furthermore, $gx'_1 = \langle \mathbf{b}, \mathbf{p} \rangle + c$ by construction of U and so P' is the cross product of $P'' = \{(\langle \mathbf{b}, \mathbf{p} \rangle + c)/g\}$ and some $P_1 \in \mathbb{R}^{d-1}$. Therefore $\mathcal{E}(P) = \mathcal{E}(P') = \mathcal{E}(P'') \cdot \mathcal{E}(P_1)$. The number of integer points in P'' is zero or one depending on the parameters and can be represented by a periodic number. Repeating the above l times, yields a $P_l \in \mathbb{R}^{d-l}$ of full dimension.

Even if P is full-dimensional, then we may in some cases still be able to write it as a cross product of two or more sets [19]. Since the dimension of each set is smaller than that of P, we can greatly reduce the computation time by calculating the number of points in each factor separately and multiplying the results afterward. Note that we also need to "multiply" the validity domains, i.e., the validity domains of the product are the intersections of the corresponding validity domains in the factors. Furthermore, if P or one of its factors is one-dimensional then it has two vertices $l(\mathbf{p}) \leq u(\mathbf{p})$ and we simply calculate $\lfloor u(\mathbf{p}) \rfloor - \lceil l(\mathbf{p}) \rceil + 1$ (again a periodic number) in each validity domain rather than using the algorithm of Section 3.

Our procedure for calculating Barvinok's decomposition into unimodular cones is an independent reimplementation of the corresponding procedure in `LattE` [12] as described in [14]. Like `LattE`, we use Shoup's implementation [34] of Lenstra Lenstra and Lovasz' basis reduction algorithm and `GMP` [16] for computing in exact long integer arithmetic. Unlike `LattE`, however, we use `PolyLib` [25] for performing polyhedral operations, since this allows us to reuse the procedures for subdividing the parameter space into validity domains [26]. The disadvantage of using `PolyLib` is that it incurs a speed penalty by insisting on maintaining the dual representation for all polytopes and by its non-optimal use of the `GMP` library. Furthermore, the worst-case running time of our algorithm is exponential in the input size, which is defined as the number of bits needed to represent the input [32]. The root cause of this behavior is that the number of elements in each periodic number is linear in the period of the Ehrhart polynomial, which is in turn bounded only by the *value* of the coefficients in the input. We are working on a different representation of periodic numbers that should alleviate this problem [37]. Our implementation is available from **`http://freshmeat.net/projects/barvinok/`**.

Two methods are often cited for counting the number of points in a parameterized polytope: Clauss and Loechner (1998) [11] and Pugh (1994) [29]. The main differences and similarities between our technique and the first have already been highlighted. The technique of Pugh consists of a set of simplification rules and the application of a set of standard summation formulas for some base cases. If he cannot find an explicit formula, he resorts to "splintering" the polytope. In contrast to our technique and that of Claus and Loechner, his technique does not appear to have been implemented yet.

Although `LattE` initially only counted the number of points in non-parameterized polytopes, it has been extended to compute Ehrhart *series* by what the authors call the ho-

[2]A unimodular matrix is an integer matrix with determinant 1 or -1.

program	nr. of polytopes	not handled by interpolation	exec. time interpolation	exec. time Barvinok
vpenta	3513	0	126.18s	58.95s
mxm	70	0	6.49s	1.24s
liv18	3052	6	255.37s*	47.60s
cholesky	77	0	3.48s	1.33s
jacobi	167	4	22.36s*	2.74s
gauss-jordan	238	0	12.64s	4.06s
tomcatv	5091	36	336.86s*	87.77s
total	12210	46	763.38s	203.69s

Table 1: Number of polytopes that are counted while calculating forward reuse distances for the different programs. For 46 polytopes, the interpolating method cannot find the solution. Furthermore, the presented analytical calculation is about 3.7 times faster than the interpolation method.

mogenized Barvinok algorithm [13]. An Ehrhart series is a formal power series that is closely related to an Ehrhart polynomial; they are in fact polynomially interconvertible. The main difference is that in an Ehrhart series the number of points in the dilatation pP is equal to the *coefficient* of the term t^p. It does not appear to be as directly usable in our context, but it has interesting mathematical properties. They apparently only handle simple dilatations with a single parameter, which means that they cannot handle problems with parameter spaces that need to be partitioned into validity domains.

Recently some advances have been made towards automata-based counting [7, 28]. Although these techniques handle a larger class of problems (solutions to Presburger formulas), they do not support symbolic parameters.

6. CONCLUSION

In this work, we have presented a new counting method targeted at arbitrary parameterized polytopes. It consists in computing Ehrhart polynomials using a novel analytical technique, based on Barvinok's decomposition. In contrast to the previously known interpolation method, our method always produces a solution, eliminating any degeneracy problem. We have used this method for computing reuse distances, which can be represented as the number of integer points in unions of parameterized polytopes. Communicating this information to the processor through cache hints leads to an improved cache replacement policy.

Although our current implementation is already faster than the previous one, we still expect long computation times for problems that result in large periods. We are therefore working on an alternative representation of periodic numbers [37]. Furthermore, extending our reuse distance formalism to caches with a longer line size and limited associativity may lead to Presburger formulas that cannot be simplified to unions of polytopes. In particular, we will need to handle sets with descriptions containing existentially quantified variables. Counting the number of points in such sets is equivalent to counting the number of points in integer projections of parameterized polytopes. To the best of our knowledge, this problem has only been solved for special cases (e.g., projecting out a single dimension [33]) or, for the non-parameterized case, with a mathematical algorithm that does not appear to be practically implementable [3].

7. REFERENCES

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

[2] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In *34th Annual Symposium on Foundations of Computer Science*, pages 566–572. IEEE, November 1993.

[3] Alexander Barvinok and Kevin Woods. Short rational generating functions for lattice point problems, November 2002. http://arxiv.org/abs/math.CO/0211146.

[4] Kristof Beyls and Erik D'Hollander. Reuse distance as a metric for cache behavior. In *IASTED conference on Parallel and Distributed Computing and Systems 2001 (PDCS01)*, pages 617–662, 2001.

[5] Kristof Beyls and Erik D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, pages 265–274, 2002.

[6] A. J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, University of Leiden, The Netherlands, 1996.

[7] B. Boigelot and L. Latour. Counting the solutions of Presburger equations without enumerating them. In *Proc. 6th International Conference on Implementations and Applications of Automata (Revised Papers), volume 2494, Lecture Notes in Computer Science*, pages 40–51, July 2001.

[8] V. Braberman, D. Garbervetsky, and S. Yovine. On synthesizing parametric specifications of dynamic memory utilization. Technical report, October 2003.

[9] Michel Brion and Michèle Vergne. Residue formulae, vector partition functions and lattice points in rational polytopes. *J. Amer. Math. Soc.*, 10:797–833, 1997.

[10] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 286–297. ACM Press, 2001.

[11] Philippe Clauss and Vincent Loechner. Parametric Analysis of Polyhedral Iteration Spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, July 1998.

[12] J. A. De Loera, D. Haws, R. Hemmecke, P. Huggins, J. Tauzer, and R. Yoshida. A user's guide for latte v1.1, November 2003. software package `LattE` is available at `http://www.math.ucdavis.edu/~latte/`.

[13] Jesus De Loera, David Haws, Raymond Hemmecke, Peter Huggins, Bernd Sturmfels, and Ruriko Yoshida. Short rational functions for toric algebra and applications, July 2003. http://arxiv.org/abs/math.CO/0307350.

[14] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes, March 2003. http://www.math.ucdavis.edu/˜latte/theory.html.

[15] E. Ehrhart. Polynomes arithmétiques et méthode des polyèdres en combinatoire. *International Series of Numerical Mathematics*, 35, 1977.

[16] Free Software Foundation, Inc. GMP. Available from `ftp://ftp.gnu.org/gnu/gmp`.

[17] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[18] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[19] N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *Static Analysis Symposium, SAS'03*, San Diego, June 2003. LNCS 2694, Springer Verlag.

[20] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transaction on Computers*, 48(10):1009–1024, oct 1999.

[21] P. Jain, S. Devadas, D. Engels, and L. Rudolph. Software-assisted replacement mechanisms for embedded systems. In *International Conference on Computer Aided Design*, pages 119–126, nov 2001.

[22] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The Omega library. Technical report, University of Maryland, November 1996.

[23] Alvin Lebeck and David Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, jan 1997.

[24] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. MRTC report, Dept. of Computer Science and Engineering, Mälardalen University, April 2003. `http://www.mrtc.mdh.se/publ.php3?id=0531`.

[25] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France, March 1999.

[26] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, December 1997.

[27] B. Meister. Using periodics in integer polyhedral problems. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France, October 2003.

[28] Erin Parker and Siddhartha Chatterjee. An automata-theoretic algorithm for counting solutions to Presburger formulas. 2004. Accepted to Compiler Construction 2004.

[29] 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.

[30] Edwin Rijpkema, Ed Deprettere, and Bart Kienhuis. Compilation from matlab to process networks. In *Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99)*, 1999.

[31] Jesús Sánchez and Antonio González. Fast, accurate and flexible data locality analysis. In *PACT '98*, pages 124–129, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[32] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[33] Rachid Seghir. Dénombrement des point entiers de l'union et de l'image des polyédres paramétrés. Master's thesis, ICPS, Université Louis Pasteur de Strasbourg, France, June 2002.

[34] Victor Shoup. NTL. Available from `http://www.shoup.net/ntl/`.

[35] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. A compile time based approach for solving out-of-order communication in Kahn Process Networks. In *IEEE 13th International Conference on Aplication-specific Systems, Architectures and Processors (ASAP'2002)*, July 2002.

[36] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170, 1997.

[37] Sven Verdoolaege, Kristof Beyls, Maurice Bruynooghe, Rachid Seghir, and Vincent Loechner. Analytical computation of Ehrhart polynomials and its applications for embedded systems. Report CW 376, Department of Computer Science, K.U.Leuven, Leuven, Belgium, jan 2004.

[38] Fubo Zhang. *The FPT Parallel Programming Environment*. PhD thesis, Ghent University, 1996.