An Automata-Theoretic Algorithm for Counting Solutions to Presburger Formulas

Erin Parker*[1] and Siddhartha Chatterjee[2]

[1] Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599, USA, parker@cs.unc.edu
[2] IBM T. J. Watson Research Center, 1101 Kitchawan Road, Yorktown Heights, NY 10598, USA, sc@us.ibm.com

Abstract. We present an algorithm for counting the number of integer solutions to selected free variables of a Presburger formula. We represent the Presburger formula as a deterministic finite automaton (DFA) whose accepting paths encode the standard binary representations of satisfying free variable values. We count the number of accepting paths in such a DFA to obtain the number of solutions without enumerating the actual solutions. We demonstrate our algorithm on a suite of eight problems to show that it is universal, robust, fast, and scalable.

1 Introduction

The problem of counting integer solutions to selected free variables of Presburger formulas is relevant to several applications in the analysis and transformation of scientific programs. Researchers have used Presburger formulas to model the iterations of a loop nest [1], the memory locations touched by a loop nest [2], the cache lines touched by a loop nest [2], and the memory accesses incurring cache misses [3]. The well-known Cache Miss Equations (CME) compiler framework [4] for analyzing and tuning memory behavior is based on Presburger arithmetic. Counting the number of solutions to such formulas allows us to estimate the execution time of a loop nest, to evaluate the message traffic generated by a loop nest, and to determine the cache miss rate of a loop nest, among other things. However, the counting problem is inherently difficult to solve in its full generality. Early work [2,5,6,7,8] therefore took *ad hoc* approaches to this counting problem; more recent work by William Pugh [9] and Philippe Clauss [1] provided solution methods that are more systematic and complete.

Pugh's method [9] is based on computing sums. He introduces techniques for summing a polynomial over all integer solutions of selected free variables of a Presburger formula. He expresses the number of solutions symbolically, in terms of symbolic constants that are the remaining free variables in the Presburger formula. For a given summation problem, the choice of which techniques to apply and the order in which to apply them seems to require human intelligence. To our knowledge, no software implementation of Pugh's method exists.

* Erin Parker is supported by a DOE High-Performance Computer Science Fellowship.

E. Duesterwald (Ed.): CC 2004, LNCS 2985, pp. 104–119, 2004.

Clauss' method [1] is based on the vertices of parameterized polytopes and Ehrhart (pseudo-)polynomials [10]. He models a Presburger formula as a disjoint union of rational convex polytopes, the number of integer points in which is equal to the number of integer solutions to selected free variables in the formula. Clauss' method requires a geometric preprocessing step to express the Presburger formulas as disjoint polyhedra. Clauss expresses the number of included integer points using Ehrhart polynomials and periodic numbers, where the free variables are the size parameters (or the remaining free variables in the formula). Like Pugh, Clauss produces a symbolic expression of the number of solutions. Clauss' method is implemented in the Polylib library [11].

The examples that Pugh [9] and Clauss [1] handle have two characteristics: they are smooth functions of their input arguments, and the constants and coefficients that they contain are small (typically order 10–100). Our interest in Presburger arithmetic arises from modeling cache misses, for which the formulas are distinctly non-smooth (see Fricker *et al.* [12] for examples) and contain large constants and coefficients (cache capacity and array starting addresses being typical examples). The methods of Pugh and Clauss are impractical for handling such formulas. Pugh's method splinters the domain of a non-smooth formula into many subdomains, and the choice of the applicable techniques and their order of application grows combinatorially. Clauss' method requires an Ehrhart polynomial of very high degree and an impractically large number of coefficients, and is also subject to geometric degeneracies for many problems of interest.

Ghosh *et al.* make similar observations in their CME work commenting that "... to make the framework more effective, we should be able to automatically solve or at least find the number of solutions to the CMEs in parametric form" [4]. They mention the methods of Pugh and Clauss as possibilities, but, given the intrinsic difficulty of the counting problem, do not use these methods to find the number of cache misses explicitly. Instead, they use the CMEs indirectly to derive problem-specific optimization algorithms, and compromise the generality of their framework as a result.

Because of the insufficiency of existing methods for our purposes, we considered other general approaches for counting solutions to Presburger formulas that would work well for non-smooth formulas. Our counting algorithm is fundamentally different from the methods of Pugh and Clauss: it builds on a deep connection between Presburger arithmetic and automata theory, and then converts the counting problem for solutions of the Presburger formula into a graph-theoretic path counting problem on the related deterministic finite automaton (DFA). The key difference between our algorithm and the methods of Pugh and Clauss lies in symbolic capabilities. Both Pugh and Clauss express the number of solutions as closed-form functions of symbolic constants. *Our algorithm is not symbolic.* It produces the number of solutions for a particular set of numerical values of the symbolic constants, *i.e.*, the result of evaluating Pugh's or Clauss' function at a particular point. Whether this lack of symbolic capabilities is a limitation depends on the application. For instance, scheduling applications require closed-form formulas, while actual numbers may be sufficient for load balancing.

Our counting algorithm is not the only non-symbolic approach for determining the number of integer solutions to Presburger formulas. Barvinok's algorithm [13] (with subsequent improvements by Dyer and Kannan [14]) counts the number of integer points inside a convex polyhedron of fixed dimension in polynomial time. The LattE (Lattice point Enumeration) tool [15], the first known implementation of Barvinok's algorithm, is software for the enumeration of all lattice points inside a rational convex polyhedron. To count the number of solutions to a Presburger formula, Barvinok's algorithm requires a geometric preprocessing step to express the formula as a disjoint union of polyhedra. The LattE tool does not implement the preprocessing step. Our algorithm requires no such geometric preprocessing technique. The quite recent availability of the LattE software combined with the problem of geometric preprocessing has prevented a direct comparison with our counting approach.

The following points summarize the strengths and weaknesses of our counting algorithm.

- The algorithm is *universal*, handling any problem that is expressible in Presburger arithmetic.
- The algorithm is *robust*, handling an arbitrary representation of a problem without need for geometric preprocessing.
- The algorithm is *implemented*.
- The algorithm is *fast*, as we will demonstrate in Sect. 5.
- The algorithm can handle formulas with *large coefficients and constants*, as indicated by the example problems in Sect. 5.
- The algorithm does not handle *symbolic* constants.

The remainder of this paper is structured as follows. Section 2 provides a brief review of Presburger arithmetic. Section 3 establishes the connection between Presburger arithmetic and DFAs. Section 4 presents our counting algorithm. Section 5 demonstrates the algorithm on three examples. Section 6 concludes.

2 Presburger Arithmetic

Presburger arithmetic is the first-order theory of natural numbers with addition. A Presburger formula consists of affine equality and/or inequality constraints connected via the logical operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), and the quantifiers $\forall$ (for all) and $\exists$ (there exists). It is well-known how to express certain nonlinear constraints in Presburger arithmetic, such as floors, ceilings, quotients, remainders [9], and bit-interleaving [3].

Consider Presburger formula $P(i, j, k; n) = 1 \leq i \leq n \wedge 3 \leq j \leq i \wedge j \leq k \leq 5$ with free integer variables i, j, and k and symbolic constant n,[1] and consider the sum $S(n) = \sum_{i=1}^{n} \sum_{j=3}^{i} \sum_{k=j}^{5} 1$ with symbolic constant n. We say that

[1] Although i, j, k, and n are all free variables of the formula in the terminology of logic, we distinguish between those free variables that appear as summation indices in the corresponding sum and those that do not.

formula $P(i, j, k; n)$ "describes" the sum $S(n)$ because, for any given value of n, the number of solutions to the Presburger formula is equal to the value of the sum, *i.e.*, $\forall n : S(n) = |\{(i, j, k) : P(i, j, k; n) = \mathbf{true}\}|$.

Presburger arithmetic is decidable [16]; however, the complexity of the decision procedure is superexponential in the worst case. For a sentence of length n, the bound on storage and time required is $2^{2^{2^{pn}}}$, for some constant $p > 1$ [17]. This bound is tight [18]. However, it is a worst-case bound, and prior use of Presburger arithmetic in program analysis contexts has generally been well-behaved in terms of complexity of decision and simplification algorithms.

3 The Presburger Arithmetic–DFA Connection

Our counting method exploits a fundamental connection between Presburger arithmetic and automata theory, namely, that there exists a DFA recognizing the positional binary representation of the solutions of any Presburger formula. Following standard terminology, we define a DFA M as a 5-tuple $(S, \Sigma, \delta, q_0, F)$, where S is a finite set of states, Σ is a finite set of symbols called the alphabet, $\delta : S \times \Sigma \to S$ is the transition function, $q_0 \in S$ is the start state, and $F \subseteq S$ is a set of final states. This connection is perhaps not surprising, given that DFAs can describe arithmetic on the binary representation of natural numbers. The key point to remember in transitioning from Presburger arithmetic to DFAs is that we are moving from a domain of values (natural numbers) to a domain of representations (positional binary encoding).

Büchi [19,20] originally proved that a subset of $(\{0, 1\}^n)^*$ is recognizable by a finite state automaton if and only if it is definable in WS1S (Weak Second-order Theory of One Successor). Boudet and Comon [21] build on this result. Because Presburger arithmetic can be embedded in WS1S, there is a DFA recognizing the solutions of a Presburger formula. Boudet and Comon formalize the connection between Presburger arithmetic and automata in the following theorem.

Theorem 1 (Boudet-Comon [21]). *Let $\phi \equiv Q_n x_n, \ldots, Q_1 x_1 \psi$ be a formula of Presburger arithmetic where Q_i is either $\exists$ or $\forall$ and ψ is an unquantified formula with variables $x_1, \ldots, x_n$, $y_1, \ldots, y_m$. There is a deterministic and complete automaton recognizing the solutions of ϕ with at most $O(2^{2^{2^{|\phi|}}})$ states, where $|\phi| = K(\phi) + V(\phi)$, $K(\phi)$ being the sum of the sizes of each constant in the formula ϕ, written in binary representation, and $V(\phi)$ being the number of variables in formula ϕ.*

The proof of the theorem is constructive, with the construction procedure defined by induction on the structure of ϕ. The base cases are linear equalities and inequalities, for which DFA recognizers are easy to construct [21,22,23]. Logical connectives of subformulas utilize closure properties of regular sets under intersection, union, and complementation [24]. Existential quantification is handled by projecting the alphabet and the transition function (producing a nondeterministic finite automaton) followed by determinization and state minimization. Universal quantification exploits the tautology $\forall x \phi \equiv \neg \exists x \neg \phi$.

Note that moving from Presburger formulas to DFAs does not circumvent the difficulty of counting solutions, since both the decidability of a Presburger formula and the construction of a DFA recognizing the formula have triple exponential worst-case complexity. We choose to exploit the Presburger Arithmetic-DFA connection because the complexity manifests in a better understood and manageable form. Specifically, it is the translation of universal quantifiers and negation that can cause an exponential blowup in the number of states.

The connection between Presburger arithmetic and DFAs has been widely studied in the context of verification of system properties such as safety and liveness. Several authors [22,23,25,26,27] have refined the procedure for constructing automata from Presburger formulas, extending the scope of formulas from natural numbers to integers, allowing the mixing of integer and real variables, and providing tighter bounds on the number of states of the resulting DFA. In particular, Bartzis and Bultan [22,23] present algorithms for constructing finite automata that represent integer sets satisfying linear constraints. They use MONA [28], an automata manipulation tool, to implement these algorithms. Compared to similar approaches for automata representation, the methods of Bartzis and Bultan give tighter bounds on the size of generated automata. We use the Bartzis and Bultan construction algorithms to represent an affine equality/inequality constraint as a deterministic finite automaton. Using the automata-theoretic operations of union, intersection, complementation, and projection (offered in the MONA tool's automata package), we then combine such DFA representations of the affine equality/inequality constraints constituting a Presburger formula to get the DFA representation of the formula.

4 The Counting Algorithm

We now present our algorithm for counting solutions to a Presburger formula. Section 4.1 offers an example DFA used to illustrate our algorithm. Section 4.2 describes how a DFA represents the solutions of a Presburger formula. Section 4.3 looks at treating the DFA as a directed, edge-weighted graph. Section 4.4 explains how path length is used to ensure a correct solution count. Section 4.5 presents our algorithm for counting accepting paths in a DFA. [2]

4.1 Example DFA

For illustration of our counting algorithm, consider the following running example. The Presburger formula shown in Fig. 1 describes *interior misses* [3] incurred by loop nest **L** (of Fig. 2.a). Figure 2.b shows a version of this formula simplified using the Omega Calculator [30,31]. Notice that the clauses in Fig. 2.b are not disjoint. However, this potential difficulty is not an issue during the construction of a DFA to represent the formula. In fact, the formula requires no geometric preprocessing for DFA construction, which makes our algorithm very robust.

[2] For the proofs of all lemmas and theorems in this section, see [29].

[3] An interior miss is a particular type of cache miss introduced by Chatterjee *et al.* [3].

$$\exists d : 0 \le i, j, k < 20 \ \wedge \ ((k = 0 \ \wedge \ R = 0 \ \wedge \ 4(128d) \le 800 + i + 20j < 4(128d + 1)) \ \vee \ (R = 1 \ \wedge \ 4(128d) \le i + 20k < 4(128d + 1)) \ \vee \ (R = 2 \ \wedge \ 4(128d) \le 400 + k + 20j < 4(128d + 1)) \ \vee \ (k = 19 \ \wedge \ R = 3 \ \wedge \ 4(128d) \le 800 + i + 20j < 4(128d + 1))) \ \wedge \ (\exists u, v, w, F, e : 0 \le u, v, w < 20 \ \wedge \ (u < i \ \vee \ (u = i \ \wedge \ v < j) \ \vee \ (u = i \ \wedge \ v = j \ \wedge \ w < k) \ \vee \ (u = i \ \wedge \ v = j \ \wedge \ w = k \ \wedge \ F < R)) \ \wedge \ ((w = 0 \ \wedge \ F = 0 \ \wedge \ 4(128e) \le 800 + u + 20v < 4(128e + 1)) \ \vee \ (F = 1 \ \wedge \ 4(128e) \le u + 20w < 4(128e + 1)) \ \vee \ (F = 2 \ \wedge \ 4(128e) \le 400 + w + 20v < 4(128e + 1)) \ \vee \ (w = 19 \ \wedge \ F = 3 \ \wedge \ 4(128e) \le 800 + u + 20v < 4(128e + 1))) \ \wedge \ (\neg(\exists x, y, z, G : 0 \le x, y, z < 20 \ \wedge \ (x < i \ \vee \ (x = i \ \wedge \ y < j) \ \vee \ (x = i \ \wedge \ y = j \ \wedge \ z < k) \ \vee \ (x = i \ \wedge \ y = j \ \wedge \ z = k \ \wedge \ G < R)) \ \wedge \ (u < x \ \vee \ (u = x \ \wedge \ v < y) \ \vee \ (u = x \ \wedge \ v = y \ \wedge \ w < z) \ \vee \ (u = x \ \wedge \ v = y \ \wedge \ w = z \ \wedge \ F < G)) \ \wedge \ ((z = 0 \ \wedge \ G = 0 \ \wedge \ 4(128d) \le 800 + x + 20y < 4(128d + 1)) \ \vee \ (G = 1 \ \wedge \ 4(128d) \le x + 20z < 4(128d + 1)) \ \vee \ (G = 2 \ \wedge \ 4(128d) \le 400 + z + 20y < 4(128d + 1)) \ \vee \ (z = 19 \ \wedge \ G = 3 \ \wedge \ 4(128d) \le 800 + x + 20y < 4(128d + 1)))) \ \wedge \ \neg(d = e)))$$

Fig. 1. Presburger formula describing interior misses incurred by loop nest **L** in cache set 0 (assuming a direct-mapped cache with block size 32 bytes and capacity 4K bytes, double-precision arrays adjacent in memory and linearized in column-major order).

```
L: do i = 0, 19
      do j = 0, 19
         c = Z[i,j]
         do k = 0, 19
            c = X[i,k]*Y[k,j]+c
         enddo
         Z[i,j] = c
      enddo
   enddo
```

a.

$$(1 \le i \le 4 \wedge j = 5 \wedge k = 12 \wedge R = 2) \vee$$
$$(0 \le i \le 3 \wedge j = 5 \wedge k = 12 \wedge R = 2) \vee$$
$$(4 \le i \le 7 \wedge j = 11 \wedge k = 0 \wedge R = 0) \vee$$
$$(0 \le i \le 3 \wedge j = 6 \wedge k = 0 \wedge R = 1) \vee$$
$$(5 \le i \le 8 \wedge j = 5 \wedge k = 12 \wedge R = 2)$$

b.

Fig. 2. a. Loop nest **L** performs the matrix multiplication $Z = X \cdot Y$. b. Simplified version of the Presburger formula in Fig. 1.

The DFA in Fig. 3 recognizes the solutions of the Presburger formula in Fig. 2.b. The start state of the DFA, q_0, is at node 1. The final states in set F are denoted with double circles. For this DFA there is one final state at node 10. Node 2 is a garbage state. A DFA can represent n-tuples of integers in binary notation as words over the alphabet $\{0,1\}^n$ simply by *stacking* them, using an equal length representation for each integer in the tuple. A label on any edge of the DFA in Fig. 3 is a stack of four digits, each corresponding to a free variable (i, j, k, or R). Therefore, the alphabet Σ of the DFA is

$$\left\{ \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{matrix} \right\}.$$

Borrowing terminology from hardware logic design, an `X` in a stack signifies that either a 0 or 1 is possible in that position.

4.2 Encoding Free Variable Values

Given the DFA representation of a Presburger formula, each accepting path in the DFA encodes free variable values that constitute a solution to the formula. The encoding is the standard binary representation of the integer values of the

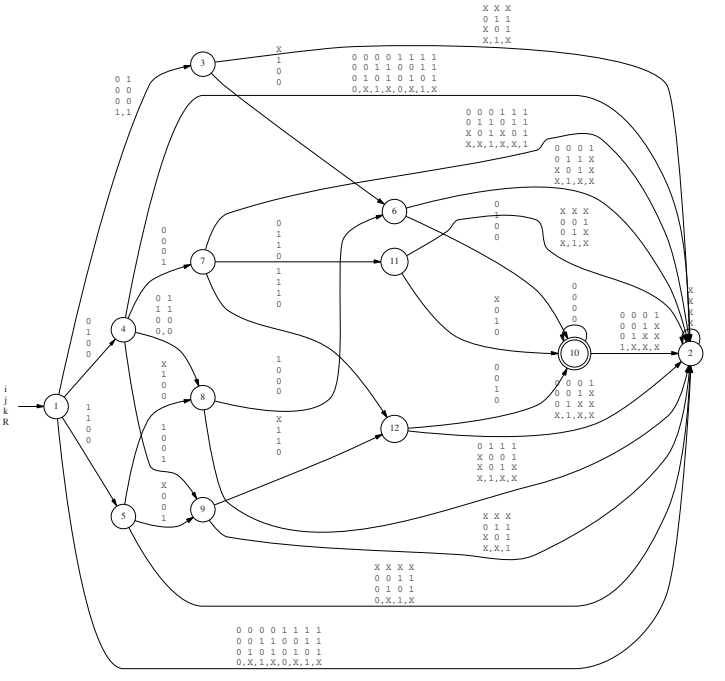

Fig. 3. DFA recognizing the solutions of the Presburger formula in Fig. 2.b.

free variables, *proceeding from least significant bit (LSB) to most significant bit (MSB)*. Formally, an encoding of a non-negative integer b is a word $b_m \dots b_0$ such that each b_i is 0 or 1 and $b = \sum_{i=0}^{m} b_i 2^i$. A tuple of non-negative integers is encoded by stacking their binary representations and padding with leading 0s to make the lengths identical.

Let a path of length i from state p to state q, denoted $P_i(p, q)$, be a string $A \in \Sigma^i$ such that $\delta^i(p, A^R) = q$, where string A^R is the *reversal* of string A. (The string reversal is a notational device to resolve the mismatch between the conventional MSB-first representation of numbers and the LSB-first consumption of the encoding by the DFA.) In the DFA of Fig. 3, one path from the state 1 to 3, $P_1(1, 3)$, is $\begin{smallmatrix}0\\0\\0\\1\end{smallmatrix}$. Path $P_4(1, 10) = \begin{smallmatrix}1000\\0101\\1100\\0010\end{smallmatrix}$, which goes through states 4, 7, and 11, is an accepting path identifyng $i = 8$, $j = 5$, $k = 12$, and $R = 2$ as a solution.

To easily identify one set of states reachable from another set of states, we extend the transition function $\delta : S \times \Sigma \to S$ to $\Delta : 2^S \to 2^S$, defined by $\Delta(S') = \bigcup_{p \in S'} \{\delta(p, a) : a \in \Sigma\}$. For the DFA in Fig. 3, $\Delta(\{6, 10, 11, 12\}) = \{2, 10\}$.

4.3 Treating the DFA as a Graph

The key to our algorithm for counting accepting paths in a DFA is to treat the DFA as a weighted, directed graph. For states p and q, let $wt(p, q) = |\{a \in \Sigma :$

$\delta(p,a) = q\}|$ be the number of alphabet symbols that cause transition from state p to state q. Given DFA $M = (S, \Sigma, \delta, q_0, F)$, define a directed, edge-weighted graph as $G(M) = (V, E, W) = (S, \{(p,q)|\exists a \in \Sigma : \delta(p,a) = q\}, \lambda(p,q).wt(p,q))$.

The problem of counting the number of solutions represented by a DFA M reduces to a path-counting problem on the graph $G(M)$. Let $N_i(q) = |P_i(q_0, q)|$ be the number of paths of length i from the vertex q_0 to vertex q. We build up N_i by induction on the path length i as follows.

Theorem 2. *For any vertex q and integer $i > 0$, $N_{i+1}(q) = \sum_{e=(p,q)\in E} N_i(p) \cdot W(e)$. If $q = q_0$ then $N_0(q) = 1$, else $N_0(q) = 0$.*

4.4 Path Length

To insure an accurate count of the accepting paths in a DFA, it is important to take into account their length. The fundamental reason for this is that the map from values to representations is one-to-many, since any representation of a value may be arbitrarily extended with leading 0s without changing the value it represents. For example, the DFA in Fig. 3 recognizes at least two different encodings for the values $i = 8$, $j = 5$, $k = 12$, and $R = 2$. Two possible paths are $P_4(1, 10) = \begin{smallmatrix}1000\\0101\\1100\\0010\end{smallmatrix}$ and $P_5(1, 10) = \begin{smallmatrix}01000\\00101\\01100\\00010\end{smallmatrix}$. Despite the fact that these *two* different encodings are recognized by the DFA, they specify only *one* solution. We want to avoid counting this solution twice, and we do so by counting all solutions identified by accepting paths of the same length (*i.e.*, by encoding the values of all free variables in the same number of bits). For any choice of length L, the number of accepting paths of that length in graph $G(M)$ is equivalent to the number of solutions to the Presburger formula represented by DFA M such that the value of each free variable comprising the solution is in the range $[0, 2^L - 1]$.

In order for counting to be feasible, the number of solutions to the Presburger formula must be finite. The Presburger formulas of interest to us are representable by bounded polytopes, where the number of formula solutions corresponds to the number of integer points in the polytopes. The bounded nature of the polytopes ensures a finite number of solutions.

Recall from Sect. 4.2 that accepting paths encode the binary representations of integer values satisfying the Presburger formula represented by the DFA (LSB to MSB). Each integer value has a unique binary representation with the exception of leading 0s. Therefore, a unique formula solution is represented in the DFA by a single accepting path that may be arbitrarily extended with 0s. Let R be the regular expression representing the set of all strings accepted by a DFA that recognizes a Presburger formula P. In order for P to have a finite set of solutions (in the value domain), the Kleene star operator can appear only in limited positions in R.

Lemma 1. *Let DFA M recognize Presburger formula P, and let regular expression R represent all accepting paths of M. Formula P has a finite number of solutions if and only if R is of the form 0^*S, where S is a regular expression free of the Kleene star operator.*

Lemma 1 gives a property of accepting DFA paths that is critical to our counting algorithm, and it relates the property to the finiteness of Presburger formula solution counts. *Note that this property does not apply to non-accepting paths.* For example, consider the DFA in Fig. 3, which recognizes a Presburger formula with a finite number of solutions. The non-accepting path that goes through nodes 1, 5, and 2 violates the 0^*S pattern, since the self-loop at the beginning of the path arbitrarily extends it with 0s or 1s.

Let V_i be the set of vertices reachable from vertex q_0 via paths of length i. To start, q_0 is the only vertex reachable from itself with path length 0 (*i.e.*, $V_0 = \{q_0\}$). In general, the set of vertices reachable from vertex q_0 with path length i is $V_i = \Delta(V_{i-1})$.

The following theorem establishes that when $V_i = V_{i-1}$, the sets of vertices reachable from vertex q_0 via paths of length $k \geq i$ are identical. In other words, the set of vertices reachable from the starting vertex with paths of a certain length reaches a steady state.

Theorem 3. *Given a DFA M corresponding to a Presburger formula with a finite number of solutions, if $V_i = V_{i-1}$, then $V_k = V_{i-1}$, $\forall k \geq i$.*

Now we want to relate the condition for the set of vertices reaching a steady state to the number of accepting paths. The following theorem shows that when $V_i = V_{i-1}$, the number of paths of length i from q_0 to accepting vertices is the same as the number of such paths of length $i - 1$.

Theorem 4. *Given a DFA M corresponding to a Presburger formula with a finite number of solutions, if $V_i = V_{i-1}$, then $\sum_{q \in F \cap V_i} N_i(q) = \sum_{q \in F \cap V_{i-1}} N_{i-1}(q)$.*

Given Theorems 3 and 4, we know that when the set of vertices reaches a steady state ($V_k = V_{i-1}$, $\forall k \geq i$), the number of accepting paths does as well ($\sum_{q \in F} N_k(q) = \sum_{q \in F} N_{i-1}(q)$, $\forall k \geq i$). Notice that set V_k can contain garbage states, but the paths reaching steady state are only the accepting paths. Our counting algorithm terminates when $V_i = V_{i-1}$ because we can be sure that the number of accepting paths in the DFA has converged.

4.5 The Counting Algorithm

The algorithm shown in Fig. 4 counts the number of accepting paths of length L in a directed, edge-weighted graph $G(M)$. We assume a representation of DFA $M = (S, \Sigma, \delta, q_0, F)$ that includes the following pieces of information.

1. `states`, a list of all states in the DFA;
2. `final`, a flag for each state p indicating if $p \in F$;
3. `to`, for each state p a list of the states q such that there exists a transition from p to q; and
4. `trans`, a transition table such that an element p, q is the list of alphabet symbols causing transition from state p to state q (note that in general this table is quite sparse).

Algorithm 1. Counting solutions to Presburger formula.
Input: DFA $M = (S, \Sigma, \delta, q_0, F)$ corresponding to Presburger formula.
Output: Path length L, number of solutions to original Presburger formula such that the value of each free variable is in the range $[0, 2^L - 1]$.
Method:

```
1   Construct the graph G(M) = (V, E, W) from M.
2   V_0 ← {q_0}
3   i ← 0
4   repeat
5       i ← i + 1
6       V_i ← ∅
7       for all q ∈ V : ∃p ∈ V_{i-1} ∧ (p, q) ∈ E do
8           V_i ← V_i ∪ q
9           Calculate N_i(q) using Theorem 2.
10      enddo
11  until V_i = V_{i-1}
12  L ← i − 1
13  return L, Σ_{q∈F} N_L(q)
```

Fig. 4. Algorithm 1 counts solutions to a Presburger formula.

Notice that we are counting the number of accepting paths in a directed graph *without enumerating each accepting path.* Therefore, the cost of our counting algorithm is sublinear in the number of solutions. The cost of line 1 is $O(|V|+|E|)$. The cost of lines 7–10 is $O(|V|^2)$, at worst. The cost of line 13 is $O(|V|)$. Finally, the worst-case bound on the complexity of the entire algorithm is $O(|V|^3)$. Empirical results in Sect. 5 suggest that the complexity is actually subquadratic in $|V|$. For a more detailed breakdown of Algorithm 1's complexity, see [29].

5 Examples

Our original experiment considered eight example problems. Because of space constraints, this section gives results of using our counting algorithm on three of those examples [4] (see [29] for the complete set of results). Examples 1 and 2 come from related work [1,2] and are handled by Clauss [1], while Pugh [9] handles Example 1. Example 3 comes from [3] and is included to show the generality and robustness of our algorithm. We use the DFA-construction algorithms of Bartzis and Bultan [22,23] to construct the DFA representation of the Presburger formula in each example. All running times were collected on a 450MHz Sparc Ultra 60. *Note that Pugh and Clauss give no running times for the application of their techniques, so we cannot compare our running times to theirs.*

[4] All three examples require geometric preprocessing to express them as disjoint unions of polyhedra, which prevents us from comparing our counting method with the LattE tool's [15] implementation of Barvinok's algorithm (see Sect. 1).

a.

n	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
4	4	64	6	0.041	3
5	5	65	7	0.045	
6	6	65	7	0.045	
7	7	67	7	0.044	
8	8	65	8	0.050	4
9	9	70	9	0.054	
10	10	67	9	0.055	
11	11	71	9	0.052	
12	12	69	9	0.053	
13	13	72	10	0.063	
14	14	70	10	0.064	
15	15	72	9	0.051	
16	16	69	10	0.062	5
17	32	85	21	0.110	
18	36	75	22	0.110	
19	38	92	20	0.103	
20	40	90	20	0.101	
21	42	91	22	0.111	
22	44	91	22	0.107	5
23	46	91	19	0.100	
24	48	86	19	0.098	
25	50	93	23	0.115	
26	52	86	23	0.114	
27	54	92	21	0.105	
28	56	90	21	0.130	
29	58	92	23	0.114	
30	60	91	23	0.115	
31	62	92	18	0.091	
32	64	76	19	0.095	6
33	97	104	35	0.255	
34	102	88	35	0.243	
35	105	109	32	0.227	
36	108	101	32	0.228	
100	700	148	50	0.475	7
500	16,000	212	76	0.826	9
1000	63,000	243	96	1.112	10
10,000	6,250,000	770	151	2.368	14

b.

μ_a	n	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
0	100	625	23	19	0.142	10
	1000	62,500	3604	30	0.188	16
21,324	100	626	35	28	0.179	10
	1000	62,501	3526	46	0.365	16

c.

$^*m$, p	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
10	902	3	20	0.152	5
100	611,252	6	48	0.608	8
1000	586,087,502	19	81	1.073	11
10,000	583,608,375,002	205	115	1.975	14

* $n = m + 1$

d.

μ_X	# of formula solutions	DFA time (ms)	# of DFA states	count time (ms)	L
81,920	325	978	109	1.770	7
80,000	225	929	110	1.660	8
85,440	250	800	126	2.125	

Fig. 5. Results of using our counting algorithm on Examples 1a (a.), 1b (b.), 2(c.), and 3(d.).

We use the following notation in the examples below. The formula P_i pertains to Example i. The notation $(\Sigma : G : x)$ indicates a guarded sum: if G is true, the value of the expression is x, otherwise it is 0. In the tables below, results include the number of solutions to the Presburger formula with n instantiated, the time required for DFA construction (given in milliseconds), the number of states in the DFA representation of the formula, the time required to count the number of accepting paths in the DFA (given in milliseconds), and the length of accepting DFA paths L. Horizontal lines in the tables indicate a change in L.

Example 1. Calculate the number of distinct cache lines touched by the loop nest in Fig. 6.a, which performs Successive Over-Relaxation. Pugh chooses a simple way of mapping array elements to cache lines, suggesting that array element $a[i, j]$ maps to cache line $[\lfloor (i-1)/16 \rfloor, j]$. Clauss does the same. We acknowledge that this mapping relation is simplistic, but use it to get our first set of results. Later we extend this example to consider a more interesting mapping relation.

The problem that Pugh and Clauss consider is described by the following Presburger formula,

$P_{1a}(x, y; n) = \exists i, j : 2 \le i \le n-1 \wedge 2 \le j \le n-1 \wedge ((16x \le i-1 < 16(x+1) \wedge y = j) \vee$
$(16x \le (i-1)-1 < 16(x+1) \ \wedge \ y = j) \ \vee \ (16x \le (i+1)-1 < 16(x+1) \ \wedge \ y = j)$
$\vee \ (16x \le i-1 < 16(x+1) \ \wedge \ y = j-1) \ \vee \ (16x \le i-1 < 16(x+1) \ \wedge \ y = j+1))$.

The table in Fig. 5.b gives the results of using our algorithm for 37 values of n. We consider $n = 4$ to 36 because Clauss does in [1]. We consider $n = 500$ because Pugh does in [9]. We also consider three other values of n. Pugh expresses the number of solutions as $(\Sigma : 3 \le n : n(1 + \lfloor (n-2)/16 \rfloor)) + (\Sigma :$

```
do i = 2, n-1
   do j = 2, n-1
      a[i,j] = (2*a[i,j]+a[i-1,j]
               + a[i+1,j]+a[i,j-1]
               + a[i,j+1])/6
   enddo
enddo
```

a.

```
do i = 0, n
   do j = 0, i+m/2
      do k = 0, i-n+p
         a[i,j,k] = a[i,j,k-1]+
                    a[k,j,k]*a[i,k,k]
      enddo
   enddo
enddo
```

b.

Fig. 6. Loop nests for the problems in Examples 1 (a.) and 2 (b.).

$n \bmod 16 = 1 \wedge n \geq 17 : n-2$) Clauss expresses the number of solutions as $(\Sigma : 1 \leq n : \frac{1}{16}n^2 + [\frac{15}{16}, \frac{7}{8}, \frac{13}{16}, \frac{3}{4}, \frac{11}{16}, \frac{5}{8}, \frac{9}{16}, \frac{1}{2}, \frac{7}{16}, \frac{3}{8}, \frac{5}{16}, \frac{1}{4}, \frac{3}{16}, \frac{1}{8}, \frac{1}{16}, 0]n + [-2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])$. We have verified that the solution counts given by our algorithm match those given by both Pugh and Clauss.

Now we extend this example to consider a more interesting mapping relation. An array element $a[i, j]$ maps to cache line $c = \lfloor(\mu_a + \beta((i-1) + n(j-1)))/B\rfloor$, where μ_a is the starting address of array a in memory, β is the array element size in bytes, and B is the blocksize of the cache in bytes. Let $\beta = 4$ and $B = 64$. This problem is described by the following first-order logic formula,

$P_{1b}(c; \mu_a, n) = \exists i, j : 2 \leq i \leq n-1 \ \wedge \ 2 \leq j \leq n-1 \ \wedge (64c \leq \mu_a + 4(i-1+n(j-1)) < 64(c+1) \ \vee \ 64c \leq \mu_a + 4((i-1)-1+n(j-1)) < 64(c+1) \ \vee 64c \leq \mu_a + 4((i+1)-1+n(j-1)) < 64(c+1) \ \vee \ 64c \leq \mu_a + 4(i-1+n((j-1)-1)) < 64(c+1) \ \vee 64c \leq \mu_a + 4(i-1+n((j+1)-1)) < 64(c+1))$.

Note that formula $P_{1b}(c; \mu_a, n)$ is not a Presburger formula unless the value of n is instantiated, as $n \cdot j$ is not a term of Presburger arithmetic. Speaking of his simple way of mapping array elements to cache lines in [9], Pugh suggests, "We could also assume more general mappings, in which the cache lines can wrap from one row to another and in which we don't know the alignment of the first element of the array with the cache lines." Such a mapping is exactly what we have above, and the formulation of the general mapping is not expressible in Presburger arithmetic for symbolic n. Therefore, Pugh would not be able to handle such a mapping without instantiating n as we have done.

We consider a version of formula $P_{1b}(c; \mu_a, n)$ simplified using the Omega Calculator [30,31]. The table in Fig. 5.c gives the results of using our algorithm for $\mu_a \in \{0, 21324\}$ and $n \in \{100, 1000\}$. Note that starting address $\mu_a = 0$ aligns with a cache line boundary and $\mu_a = 21,324$ does not, accounting for the different solution counts. Also note that solution counts differ between formulas P_{1a} and P_{1b}, with counts for the latter formula being no larger than the former. This is as expected, being the result of cache lines wrapping across rows of the array.

Example 2. Calculate the number of flops for the loop nest in Fig. 6.b. More precisely, Clauss calculates the number of iterations in the loop nest. This problem is described by the Presburger formula $P_3(i, j, k; m, n, p) = 0 \leq i \leq n \ \wedge \ 0 \leq$

p	1				2				3				4			
m	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)
1	29	3	9	0.069	56	3	10	0.080	90	3	11	0.090	130	3	12	0.096
2	32	3	9	0.068	62	3	13	0.168	100	3	12	0.093	145	3	15	0.177
3	32	3	9	0.069	62	3	13	0.166	100	3	12	0.094	145	3	15	0.176
4	35	3	9	0.071	68	3	12	0.093	110	3	13	0.168	160	3	13	0.096
5	35	3	9	0.072	68	3	12	0.091	110	3	13	0.167	160	3	13	0.098
6	38	3	9	0.068	74	3	13	0.168	120	3	12	0.091	175	3	15	0.179
7	38	3	9	0.070	74	3	13	0.167	120	3	12	0.091	175	3	15	0.181
8	41	3	9	0.069	80	3	12	0.095	130	3	13	0.168	190	3	14	0.173

p	5				6				7				8			
m	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)	# of solns	DFA (ms)	# of states	count (ms)
1	175	3	13	0.164	224	2	13	0.098	276	3	15	0.177	330	2	14	0.173
2	196	3	14	0.170	252	3	17	0.127	312	3	16	0.181	375	3	18	0.129
3	196	3	14	0.171	252	3	17	0.124	312	3	16	0.183	375	3	18	0.130
4	217	3	15	0.177	280	3	16	0.185	348	3	17	0.124	420	3	16	0.180
5	217	3	15	0.178	280	3	16	0.186	348	3	17	0.125	420	3	16	0.178
6	238	3	14	0.172	308	3	17	0.129	384	3	15	0.278	465	3	18	0.127
7	238	3	14	0.171	308	3	17	0.127	384	3	15	0.181	465	3	18	0.132
8	259	3	15	0.182	336	3	15	0.177	420	3	17	0.128	510	3	17	0.125

Fig. 7. Results of using our counting algorithm on Presburger formula $P_2(i, j, k; m, n, p)$ for $m = 1$ to 8, $n = 9$, and $p = 1$ to 8. In all cases, the path length L is 4.

$j \leq i + m/2 \wedge\ 0 \leq k \leq i - n + p$. We consider $m = 1$ to 8, $n = 9$, and $p = 1$ to 8 because Clauss does in [1], and show those results in Fig. 7. In the table in Fig. 5.d, we consider larger values of m, n, and p. Clauss expresses the number of solutions as $(\Sigma : n > p : \frac{1}{2}np^2 + \frac{3}{2}np + n + \frac{1}{4}mp^2 + \frac{3}{4}mp + \frac{1}{2}m - \frac{1}{6}p^3 + [-\frac{1}{4}, 0]_m p^2 + [\frac{5}{12}, \frac{7}{6}]_m p + [\frac{1}{2}, 1]_m)$. We have verified that our solution counts match those of Clauss.

Example 3. Consider Presburger formula $P_3(j_1, j_2, s, d; \mu_X)$ in Fig. 8. This formula represents the interior misses incurred by array X due to interference from array A during a matrix-vector multiplication. See [3] for a more detailed description of the matrix-vector multiply loop nest and the execution parameters. In the formula, μ_X is a symbolic constant and it represents the starting address of array X in memory. The table in Fig. 5.e gives the results of using our algorithm for three values of μ_X. We have verified that our solution counts match the interior miss counts reported by a (specially-written) cache simulator. Example 3 demonstrates our claim that our algorithm can handle non-smooth formulas. The other three examples are quite continuous, but this formula is very "spikey" for various values of μ_X (as Fig. 2 in Fricker *et al.* [12] shows).

Observations. The following are five noteworthy observations on the results.

1. The number of DFA states does not grow significantly as the values of the size constants increase, showing good scalability.
2. For all 160 DFAs considered in the original experiment (see [29]), the curve $y = 0.0025x^{1.4}$ fits the plot of the time required to count all accepting paths against the number of DFA states, illustrating a subquadratic relationship between the number of states and counting time. Notice that counting time is markedly sublinear in the number of solutions.
3. The running times show that the counting algorithm is far less expensive that the DFA-construction algorithms of Bartzis and Bultan [22,23]. However, both are clearly quite fast.

$$(j_2 = 0 \ \wedge \ (\exists \alpha : 1 \leq j_1 \leq 99 \ \wedge \ 0 \leq s \leq 255 \ \wedge \ \alpha < d \ \wedge \ 32s + 8192d \leq \mu_X \ \wedge \ 800j_1 + 8192d \leq 792 + \mu_X + 8192\alpha$$
$$\wedge \ \mu_X \leq 31 + 32s + 8192d \ \wedge \ s + 256\alpha < 25j_1)) \ \vee \ (\exists \alpha : 1 \leq j_1 \leq 99 \ \wedge \ 0 \leq s \leq 255 \ \wedge \ 1 \leq j_2 \ \wedge$$
$$925 + 100j_1 + j_2 \leq 1024d + 4s \ \wedge \ 8192d + 32s \leq \mu_X + 8j_2 \ \wedge \ \mu_X - 8j_2 \leq 7 + 8192d + 32s \ \wedge$$
$$100j_1 + j_2 \leq 99 + 4s + 1024\alpha \ \wedge \ s + 256\alpha < 25j_1) \ \vee \ (\exists \alpha : 0 \leq j_1 \leq 99 \ \wedge \ 0 \leq s \leq 255 \ \wedge \ j_2 \leq 99 \ \wedge$$
$$25j_1 \leq s + 256\alpha \ \wedge \ 8192d + 32s \leq \mu_X + 8j_2 \ \wedge \ \mu_X + 8j_2 \leq 7 + 8192d + 32s \ \wedge \ 4s + 1024\alpha < 100j_1 + j_2$$
$$\wedge \ 256 + 25j_1 \leq 256d - s) \ \vee \ (\exists \alpha : 0 \leq j_1 \leq 99 \ \wedge \ 0 \leq j_2 \leq 99 \ \wedge \ 0 \leq s \leq 255 \ \wedge \ 8192d + 32s \leq \mu_X - 8j_2 \ \wedge$$
$$\mu_X + 8j_2 \leq 31 + 8192d + 32s \ \wedge \ 1021 + 100j_1 + j_2 \leq 1024d + 4s \ \wedge \ 100j_1 + j_2 \leq 3 + 4s + 1024\alpha \ \wedge$$
$$4s + 1024\alpha \leq 100j_1 + j_2) \ \vee \ (j_2 = 0 \ \wedge \ (\exists \alpha : 1 \leq j_1 \leq 99 \ \wedge \ 0 \leq s \leq 255 \ \wedge \ 257 + s + 256d \leq 25j_1 \ \wedge$$
$$32s + 8192d \leq \mu_X \ \wedge \ 800j_1 + 8192d \leq 792 - \mu_X + 8192\alpha \ \wedge \ \mu_X \leq 31 - 32s + 8192d \ \wedge \ s + 256\alpha < 25j_1))$$
$$\vee \ (\exists \alpha : 1 \leq j_1 \leq 99 \ \wedge \ 0 \leq s \leq 255 \ \wedge \ 1 \leq j_2 \ \wedge \ 257 + 256d + s \leq 25j_1 \ \wedge \ 8192d + 32s \leq \mu_X + 8j_2 \ \wedge$$
$$\mu_X + 8j_2 \leq 7 + 8192d + 32s \ \wedge \ 100j_1 + j_2 \leq 99 + 4s + 1024\alpha \ \wedge \ s + 256\alpha < 25j_1) \ \vee \ (\exists \alpha : 0 \leq j_1 \leq 99 \ \wedge$$
$$0 \leq s \leq 255 \ \wedge \ j_2 \leq 99 \ \wedge \ 25j_1 \leq s + 256\alpha \ \wedge \ 8192d + 32s \leq \mu_X + 8j_2 \ \wedge \ \mu_X - 8j_2 \leq 7 + 8192d + 32s \ \wedge$$
$$4s + 1024\alpha < 100j_1 - j_2 \ \wedge \ 1025 + 1024d + 4s \leq 100j_1 + j_2) \ \vee \ (\exists \alpha : 0 \leq j_1 \leq 99 \ \wedge \ 0 \leq j_2 \leq 99 \ \wedge$$
$$0 \leq s \leq 255 \ \wedge \ 8192d \mid 32s \leq \mu_X \mid 8j_2 \ \wedge \ \mu_X \mid 8j_2 \leq 31 \mid 8192d \mid 32s \ \wedge \ 1024 \mid 1024d \mid 4s \leq 100j_1 \mid j_2$$
$$\wedge \ 100j_1 + j_2 \leq 3 + 4s + 1024\alpha \ \wedge \ 4s + 1024\alpha \leq 100j_1 + j_2)$$

Fig. 8. Presburger formula $P_3(j_1, j_2, s, d; \mu_X)$ describing interior misses on array X due to interference from array A during matrix-vector multiply.

4. Only formulas $P_{1b}(c; \mu_a, n)$ and $P_3(j_1, j_2, s, d; \mu_X)$ require simplification before our counting method is applied. For formula P_{1b}, the total amount of time to simplify all instantiations is less than 0.5 seconds. For formula P_3, the total amount of time to simplify all instantiations is less than 0.3 seconds. The Omega Calculator [30,31] is used for all simplifications.
5. The examples presented here contain large values (order 10,000). The fast running times demonstrate that our counting method can handle such values.

6 Conclusions

We have presented an automata-theoretic algorithm for counting integer solutions to selected free variables of a Presburger formula. Demonstration of our counting algorithm on three examples shows that it is robust, universal, fast, and scalable. The time required to count all accepting paths scales well with the number of DFA states and solutions.

The polyhedral methods of Pugh and Clauss and our automata-theoretic method have complementary strengths. While the formulas in Figs. 1 and 2.b would produce the same DFA in the end, the DFA construction time for the formula of Fig. 1 would be prohibitively large. We therefore routinely simplify complicated Presburger formulas such as the one in Fig. 1, P_{1b}, and P_3 as much as possible using the Omega Calculator before constructing the DFAs and counting solutions. On the other hand, by instantiating the symbolic parameters, our counting algorithm can handle non-smooth formulas that are too difficult to handle symbolically using polyhedral methods. Therefore, in cases where the symbolic methods of Pugh and Clauss can be used simply and quickly, they should be. For cases which are difficult or impossible for the methods of Pugh and Clauss to handle, our counting algorithm is a nice alternative. Our algorithm could also be effectively used as a subroutine in Clauss' method to count the number of solutions for the necessary number of small-sized polyhedra.

References

1. Clauss, P.: Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In: Proceedings of the 1996 ACM International Conference on Supercomputing. (1996) 278–285
2. Ferrante, J., Sarkar, V., Thrash, W.: On estimating and enhancing cache effectiveness. In Banerjee, U., et al., eds.: Proceedings of the Fourth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Volume 589 of Lecture Notes in Computer Science., Springer (1991) 328–343
3. Chatterjee, S., Parker, E., Hanlon, P.J., Lebeck, A.R.: Exact analysis of the cache behavior of nested loops. In: Proceedings of 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation. (2001) 286–297
4. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems **21** (1999) 703–746
5. Haghighat, M., Polychronopoulos, C.: A basis for parallelization, optimization and scheduling of programs. Technical Report 1317, CSRD, University of Illinois (1993)
6. Haghighat, M., Polychronopoulos, C.: Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In Banerjee, U., et al., eds.: Proceedings of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing. Volume 768 of Lecture Notes in Computer Science., Springer (1993) 567–585
7. Tawbi, N.: Estimation of nested loop exection time by integer arithmetics in convex polyhedra. In: Proceedings of the Eighth International Parallel Processing Symposium. (1994) 217–221
8. Tawbi, N., Feautrier, P.: Processor allocation and loop scheduling on multiprocessor computers. In: Proceedings of the 1992 ACM International Conference on Supercomputing. (1992) 63–71
9. Pugh, W.: Counting solutions to Presburger formulas: How and why. In: Proceedings of 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation. (1994) 121–134
10. Ehrhart, E.: Polynômes arithmétiques et Méthode des Polyédres en Combinatoire. Volume 35 of International Series of Numerical Mathematics. Birkhäuser Verlag, Basel/Stuttgart (1977)
11. Loechner, V.: PolyLib: A Library for Manipulating Parameterized Polyhedra. (1999)
12. Fricker, C., Temam, O., Jalby, W.: Influence of cross-interference on blocked loops: A case study with matrix-vector multiply. ACM Transactions on Programming Languages and Systems **17** (1995) 561–575
13. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Mathematics of Operations Research **19** (1994) 769–779
14. Dyer, M., Kannan, R.: On Barvinok's algorithm for counting lattic points in fixed dimension. Mathematics of Operations Research **22** (1997) 545–549
15. Loera, J.A.D., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes. http://www.math.ucdavis.edu/∼latte/pdf/lattE.pdf (2003)

16. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves. (1929) 92–101
17. Oppen, D.C.: A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. Journal of Computer and System Sciences **16** (1978) 323–332
18. Weispfenning, V.: Complexity and uniformity of elimination in Presburger arithmetic. In: Proceedings of the 1997 ACM SIGSAM International Symposium on Symbolic and Algebraic Computation. (1997) 48–53
19. Büchi, J.R.: Weak second-order arithmetic and finite automata. Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik **6** (1960) 66–92
20. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In Nagel, E., et al., eds.: Proceedings of the International Congress on Logic, Methodology and Philosophy of Science 1960, Stanford University Press (1962) 1–11
21. Boudet, A., Comon", H.: Diophantine equations, Presburger arithmetic and finite automata. In Kirchner, H., ed.: Proceedings of 21st International Colloquium on Trees in Algebra and Programming. Volume 1059 of Lecture Notes in Computer Science., Springer (1996) 30–43
22. Bartzis, C., Bultan, T.: Efficient symbolic representations for arithmetic constraints in verification. Technical Report 2002-16, Computer Science Department, University of California, Santa Barbara (2002)
23. Bartzis, C., Bultan, T.: Automata-based representations for arithmetic constraints in automated verification. In: Proceedings of the 7th International Conference on Implementation and Application of Automata. (2002) 282–288
24. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company (1979)
25. Boigelot, B., Wolper, P.: Representing arithmetic constraints with finite automata: An overview. In J.Stuckey, P., ed.: Proceedings of the 18th International Conference on Logic Programming. Volume 2401 of Lecture Notes in Computer Science., Springer (2002) 1–19
26. Wolper, P., Boigelot, B.: An automata-theoretic approach to Presburger arithmetic constraints (extended abstract). In Mycroft, A., ed.: Proceedings of the Second International Static Analysis Symposium. Volume 983 of Lecture Notes in Computer Science., Springer (1995) 21–32
27. Wolper, P., Boigelot, B.: On the construction of automata from linear arithmetic constraints. In Graf, S., Schwartzbach, M.I., eds.: Proceedings of the Sixth International Conference on Tools and Algorithms for Construction and Analysis of Systems. Volume 1785 of Lecture Notes in Computer Science., Springer (2000) 1–19
28. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. (2001) http://www.brics.dk/mona.
29. Parker, E., Chatterjee, S.: An automata-theoretic algorithm for counting solutions to Presburger formulas. Technical Report TR04-001, Department of Computer Science, University of North Carolina at Chapel Hill (2004)
30. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega Calculator and Library, Version 1.1.0. (1996) http://www.cs.umd.edu/projects/omega.
31. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega Library Version 1.1.0 Interface Guide. (1996) http://www.cs.umd.edu/projects/omega.