Scalable Linear Invariant Generation with Farkas' Lemma

Hongming Liu, Hongfei Fu, Zhiyong Yu, Jiaxin Song, Guoqiang Li

Scalable Linear Invariant Generation with Farkas' Lemma

HONGMING LIU, Shanghai Jiao Tong University, China
HONGFEI FU, Shanghai Jiao Tong University, China
ZHIYONG YU, Shanghai Jiao Tong University, China
JIAXIN SONG, Shanghai Jiao Tong University, China
GUOQIANG LI, Shanghai Jiao Tong University, China

Invariant generation is the classical problem to automatically generate invariants to aid the formal analysis of programs. In this work, we consider the linear-invariant-generation problem over affine programs (i.e., programs with affine guards and updates). In the literature, the only known sound and complete characterization to solve this problem is via Farkas' Lemma (FL), and has been implemented through either quantifier elimination or reasonable heuristics. Although FL-based approaches can generate highly accurate linear invariants from the completeness of FL, the main bottleneck to apply these approaches is the scalability issue caused by either non-linear constraints or combinatorial explosion. It has long been an unresolved problem to improve the scalability of FL-based approaches. In this work, we address this issue with novel algorithmic improvements. In detail, we base our approach on the only practical FL-based approach [Sankaranarayanan *et al.*, SAS 2004] that applies FL with reasonable heuristics, and develop two novel and independent improvements to leverage the scalability. The first improvement is the novel idea to generate invariants at only one program location in a single invariant-generation process, so that the invariants for each location are generated separately rather than together in a single computation. This idea naturally leads to a parallel processing that divides the invariant-generation task for all program locations by assigning the locations separately to mulitple processors. Moreover, the idea enables us to develop detailed technical improvements to further reduce the combinatorial explosion in the original work [Sankaranarayanan *et al.*, SAS 2004]. The second improvement is a segmented subsumption testing in the CNF-to-DNF expansion that allows to discover more local subsumptions in advance. We formally prove that our approach has the same accuracy as the original work, thus does not incur accuracy loss on the generated invariants. Moreover, experimental results on representative benchmarks involving non-trivial linear invariants demonstrate that our approach improves the runtime of the original work by several orders of magnitude, even if in the non-parallel scenario that sums up the execution time for all program locations. Thus, our approach constitutes the first significant improvement in FL-based approaches for linear invariant generation after almost two decades.

1 INTRODUCTION

Invariants. An assertion at a program location is called an *invariant* if it is always satisfied by the values taken by the program variables whenever the location is reached in the execution of the program. Invariants play a fundamental role in program analysis and verification as they provide over-approximation for reachable program states. Therefore, they are widely used in proving basic properties such as safety [3, 51, 56], reachability [4, 6, 10, 17, 19, 26, 57] and time-complexity analysis [13], etc. The quality of the generated invariants is measured by their accuracy, i.e., the amount of over-approximation against the actual set of reachable program states. The accuracy of the invariants is an important factor as inaccurate invariants can lead to loose results or even failure to get meaningful results for program analysis and verification.

Invariant Generation. Invariant generation is the classical problem that asks to automatically generate invariants for an input program, and has been studied for decades. Various approaches have been proposed to solve the problem, such as abstract interpretation [21, 23], constraint

Authors' addresses: Hongming Liu, Shanghai Jiao Tong University, Shanghai, China, hm-liu@sjtu.edu.cn; Hongfei Fu, Shanghai Jiao Tong University, Shanghai, China, fuhf@cs.sjtu.edu.cn; Zhiyong Yu, Shanghai Jiao Tong University, Shanghai, China, yuzhiyong18@sjtu.edu.cn; Jiaxin Song, Shanghai Jiao Tong University, Shanghai, China, sjtu_xiaosong@sjtu.edu.cn; Guoqiang Li, Shanghai Jiao Tong University, Shanghai, China, li.g@sjtu.edu.cn.

solving [14, 18, 45], recurrence analysis [31, 44, 47, 48], logical inference [29, 34, 35, 38, 52, 66], machine learning [36, 41, 77], dynamic analysis [25, 54, 68], etc.

To infer invariants at program locations directly is often infeasible, and most existing approaches generate invariants by considering a strengthened notion called *inductive invariants*. An inductive invariant at a program location is an assertion that holds for the first visit to the location and is preserved under every cyclic execution path to and from the location. Inductive invariants are guaranteed to be invariants, and the well-established method to prove that an assertion is an invariant is to find an inductive invariant that strengthens it [18, 51].

Numerical Inductive Invariants. An important category of inductive invariants is that of *numerical inductive invariants* that captures the relationship between the numerical values taken by the program variables. Numerical values are a basic aspect of programs, and many common failures of programs (such as array out-of-bound, division by zero, etc.) are closely related to numerical values. Thus, numerical inductive invariants are essential in proving numeric-critical properties. In this work, we consider automated generation of numerical inductive invariants. To be more precise, we consider algorithmic approaches for generating *linear* inductive invariants below.

A notable subclass of numerical inductive invariants is the class of linear inductive invariants. Informally, a numerical inductive invariant is linear if the invariant takes the form of a system of linear inequalities over the program variables. Linear inductive invariants are the most basic form of numerical invariant invariants, hence is important both for the academic and practical purpose. To resolve the automated generation of linear inductive invariants, we consider the method of constraint solving, as follows.

The Method of Constraint Solving. To solve the invariant-generation problem, constraint-solving based approaches usually consider the following paradigm: first establish a template with unknown parameters for the target invariant, then collect constraints from the inductive condition for invariants, and finally solve the unknown parameters in the template to get the desired invariants.

Constraint-solving based approaches for numerical invariant generation can roughly be classified by linear and polynomial invariant generation. For linear invariant generation, Farkas' Lemma provides a complete characterization of the inductive condition and has been studied in [18, 63], which was further solved by quantifier elimination [18] and several heuristics [63]. The STING invariant generator [71] implements the approach in [63], and the INVGEN invariant generator [39] integrates abstraction interpretation and the approach in [63]. Besides, an approach based on eigenvectors for a restricted class of invariants is proposed in [28]. Recently, probabilistic linear invariants have also been considered in probabilistic programs through Farkas' Lemma and Motzkin's Transposition theorem [15, 46].

For polynomial invariant generation, a variety of approaches were proposed in the literature. Complete approaches (that typically have high runtime complexity) were proposed through quantifier elimination [45] and other computer-algebra based techniques [76]. Semi-complete approaches (that have lower runtime complexity but retain completeness in restricted situations as compared with complete approaches) through Positivstellensätze have been proposed in [14]. The special case of polynomial-equality invariants was solved completely by Zariski-closure [42] and Gröbner basis [60]. Heuristics for polynomial invariant generation have also been extensively studied, such as semidefinite programming with relaxation [1, 20, 50], Lagrange interpolation [16], reduction to linear algebra [27], Hypergeometric sequences [43] and Gröbner basis [62]. Recently, an approach based on Stengle's Postivstellensatz for generating probabilistic polynomial invariants in probabilistic programs is proposed in [32].

Compared with other methods (such as abstract interpretation, machine learning, etc.), constraint solving has the advantage of a theoretical guarantee on the accuracy of the generated invariants

based on the considered numerical form of the invariants and the accuracy loss caused by the heuristics (if any), but typically require higher runtime complexity.

Our Contribution. We follow the constraint-solving method and consider automated generation of linear inductive invariants over affine programs. An affine program is an imperative program where every assignment is an affine expression over program variables and every guard condition (in e.g. conditional branches, while loops, etc.) is a propositional combination of comparison between affine expressions over program variables. Our work is based on the previous work [63] that implements FL with reasonable heuristics and is the only FL-based approach that achieves practical performance over realistic programs.

We propose two novel improvements to leverage the scalability of applying FL to linear invariant generation, leading to a novel approach that significantly improves the scalability of [63]. In [63], the scalability issue arises from two sources of combinatorial explosion: a CNF-to-DNF expansion and also a polyhedral-cone generator computation. Our first improvement is the novel idea that instead of solving the invariants at all program locations in one invariant-generation process when applying FL, our approach solves the invariants at each location separately, i.e., solving the invariants location-by-location. The idea directly leads to the potential to speed up the invariant-generation process at the target location since one does not need to generate the invariants at other locations, and naturally enables further speed up through parallel processing (that computes the invariants for every location separately over multiple processors). Moreover, the idea enables us to develop the following technical improvements to further mitigate the combinatorial explosion in the approach [63]: (i) in the CNF-to-DNF expansion, we reorder the expansion so that the target location (over which the invariants are to be generated) comes first to detect subsumptions earlier; (ii) in the polyhedral-cone generator computation, we eliminate variables unrelated to the target location in advance to speed up the generator computation. Our second improvement is the extra segmented subsumptiong testing that serves as a pre-processing for the CNF-to-DNF expansion and could detect local subsumption in advance.

We develop our approach on the abstract model of linear transitions systems [63] that capture general affine updates (with affine guards) between program locations, so that our approach is applicable to general affine programs. To complement the generality of our approach, we implement a prototype transformation from C program codes into linear transition systems to showcase the practical connection with concrete programs. We show that our approach generates exactly the same linear invariants as the approach [63] does, thus inherits the merit of high accuracy from constraint solving. Moreover, experimental results on typical benchmarks which involves non-trivial linear invariants reveal that our approach can indeed substantially improves the scalability against the state-of-the-art approach [63]. For example, for several large-scale benchmarks our approach is able to attain up to thousands of orders of magnitude speed-up when compared with [63]. As a result, our approach constitutes to our best knowledge the first significant improvement in Farkas'-Lemma based methods after almost two decades (since [63]).

2 LINEAR TRANSITION SYSTEMS AND INVARIANTS

We consider linear transition systems (LinTS's) [63] as the underlying model for invariant generation. A LinTS is composed of locations and linear transitions between locations, thus is suitable for modelling the executions of an affine program, for which a location in a LinTS corresponds to a program location (a.k.a program counter) of an affine program, and the linear transitions corresponds to the affine updates (arising from assignment statements) and guards (from if-branches and while-loops) in the program.

To present the definitions for LinTS's, we first define the basic notions of linear (in)equalities and assertions. For linear inequalities, we only consider the non-strict comparison operator $\geq$. Note that although an equality $\alpha = \beta$ can be equivalently expressed by two inequalities $\alpha \leq \beta$ and $\alpha \geq \beta$, the equalities in a LinTS are tackled directly as various optimizations could be applied to equalities. Also note that inequalities of the form $\alpha \leq \beta$ could be equivalently tranformed into $-\alpha \geq -\beta$.

Linear (In)equalities and Assertions. A *linear equality* over a set $V = \{x_1, \ldots, x_n\}$ of real-valued variables is of the form $a_1x_1 + \cdots + a_nx_n + b = 0$, where a_i's and b are real coefficients. Analogously, a *linear inequality* over V is of the form $a_1x_1 + \cdots + a_nx_n + b \geq 0$. A *linear assertion* over V is a conjunction of linear equalities and inequalities over V.

Then we present the abstract model of linear transition systems.

Linear Transition Systems. A *linear transition system* (LinTS) is a tuple $\langle X, X', L, \mathcal{T}, \ell^*, \theta\rangle$ where:

- X is a finite set of real-valued variables such that each variable $x \in X$ represents the current value of the variable, while $X' = \{x' \mid x \in X\}$ is the corresponding set of *primed variables* such that each primed variable $x' \in X'$ represents the value of the unprimed counterpart $x \in X$ in the next step of the system;
- L is a finite set of *locations* and $\ell^* \in L$ is the initial location;
- $\mathcal{T}$ is a finite set of *transitions* where each transition τ is a triple $\langle \ell, \ell', \rho\rangle$ that specifies the jump from the current location ℓ to the next location ℓ' with the guard condition ρ as a linear assertion over $X \cup X'$;
- θ is a linear assertion over the variables X that specifies the initial condition at the initial location ℓ^*.

To describe the behaviour of a LinTS, we further define the notions of valuations, configurations and their associated satisfaction relation as follows.

Valuations and Configurations. A *valuation* over a variable set V is a function $\sigma : V \to \mathbb{R}$ that assigns to each variable $x \in V$ a real value $\sigma(x)$ that corresponds to the current value held by x. In this work, we mainly consider valuations over the variable set X of a LinTS and simply abbreviate "valuation over X" as "valuation" (i.e., omitting X). Given a LinTS, a *configuration* is a pair (ℓ, σ) such that $\ell \in L$ is a location and σ is a valuation (over X), with the intuition that ℓ is the current location and σ specifies the current values for the variables in the LinTS.

The Satisfaction Relation. Given a linear assertion φ over X and a valuation σ, we write $\sigma \models \varphi$ to mean that σ satisfies φ, i.e., φ is true when one substitutes the corresponding values $\sigma(x)$ in σ to all the variables x in φ. Analogously, given two valuations σ, σ' (over X) and a linear assertion φ over $X \cup X'$, we write $\sigma, \sigma' \models \varphi$ to mean that φ is true when one substitutes every variable $x \in X$ by $\sigma(x)$ and every variable $x' \in X'$ by $\sigma'(x)$ in φ. Moreover, given two linear assertions φ, ψ over X, we write $\varphi \models \psi$ to mean that it is always the case that φ implies ψ, i.e., for every valuation σ we have that $\sigma \models \varphi$ implies $\sigma \models \psi$.

Now we describe the semantics of a LinTS.

The Semantics of LinTS's. Informally, a LinTS starts at its initial location ℓ^* with an arbitrary initial valuation σ^* such that $\sigma^* \models \theta$, constituting an initial configuration $(\ell_0, \sigma_0) = (\ell^*, \sigma^*)$; then at each step n ($n \geq 0$), given the current configuration (ℓ_n, σ_n), the LinTS determines the next configuration $(\ell_{n+1}, \sigma_{n+1})$ by first selecting a transition $\tau = \langle \ell, \ell', \rho\rangle$ such that $\ell = \ell_n$ and then choosing $(\ell_{n+1}, \sigma_{n+1})$ to be any configuration that satisfies $\ell_{n+1} = \ell'$ and $\sigma_n, \sigma_{n+1} \models \rho$. Formally, the semantics of an LinTS is specified by the notion of paths. A *path* π is a finite sequence of configurations $(\ell_0, \sigma_0) \ldots (\ell_n, \sigma_n)$ such that

- (**Initialization**) $\ell_0 = \ell^*$ and $\sigma_0 \models \theta$, and

$$X = \{x, y, t\}, L = \{\ell_0, \ell_1\}, \mathcal{T} = \{\tau_1, \tau_2\}, \tau_1 : \langle \ell_0, \ell_1, \rho_1 \rangle, \tau_2 : \langle \ell_1, \ell_0, \rho_2 \rangle, \theta : x = 0 \wedge y = 0 \wedge t = 0,$$
$$\rho_1 : \begin{bmatrix} t' - t \leq x' - x \leq 2(t' - t) \wedge \\ t' - t \leq y' - y \leq 2(t' - t) \wedge \\ 1 \leq t' - t \leq 2 \end{bmatrix}, \rho_2 : \begin{bmatrix} t' - t \leq x' - x \leq 2(t' - t) \wedge \\ -(t' - t) \leq y' - y \leq -2(t' - t) \wedge \\ 1 \leq t' - t \leq 2 \end{bmatrix}$$

Fig. 1. The LinTS for a Vagrant Robot

- (**Consecution**) for every $0 \leq k \leq n-1$, there exists a transition $\tau = \langle \ell, \ell', \rho \rangle$ satisfying $\ell = \ell_k$, $\ell' = \ell_{k+1}$ and $\sigma_k, \sigma_{k+1} \models \rho$.

Intuitively, a path starts with some legitimate initial configuration (as specified by **Initialization**) and evolves by repeatedly applying the transitions to the current configuration (as described in **Consecution**). Thus, any path $\pi = (\ell_0, \sigma_0) \dots (\ell_n, \sigma_n)$ corresponds to a possible evolution of the underlying LinTS.

Example 1. *Consider a scenario of a vagrant robot from [18]. The control of the robot works in two alternating modes modelled as two locations ℓ_0, ℓ_1. Each mode takes a time between 1 and 2 seconds to complete its task. In mode ℓ_0, the robot moves in the positive direction of both x and y, and in mode ℓ_1, it moves in the positive direction of x and the negative direction of y. Figure 1 shows the LinTS of a vagrant robot that consists of three variables x, y, t and the two locations ℓ_0, ℓ_1. The variable pair (x, y) corresponds to the current position of the robot on the two-dimensional plane, while the variable t records the amount of the elapsed time. From the LinTS, we have the following:*

- *At the start, the robot is at its initial position $(0, 0)$ (i.e., $x = 0$ and $y = 0$) at time $t = 0$ in the initial mode ℓ_0.*
- *At the location ℓ_0, the robot takes the transition τ_1. During the transition, the robot first moves for a time period $\Delta t = t' - t$ between 1 to 2 seconds (as indicated by the linear assertion $1 \leq t' - t \leq 2$ in ρ_1), for which the movement is in the positive direction of both the x- and y-axis, each with a nondeterministic distance that falls in the interval $[\Delta t, 2\Delta t]$ (as specified by the linear assertions $\Delta t \leq x' - x \leq 2\Delta t$ and $\Delta t \leq y' - y \leq 2\Delta t$ in ρ_1); then the robot changes its mode to ℓ_1.*
- *At the location ℓ_1, the robot takes the transition τ_2 and changes its mode to ℓ_0. The movement of the robot is specified by ρ_2 and similar to the situation at ℓ_0. The only difference is that the robot now moves in the positive direction along the x-axis and in the negative direction along the y-axis.*
- *The robot switches between ℓ_0 and ℓ_1 by alternately taking the transitions τ_1 and τ_2.*

A path under this LinTS is $(\ell_0, (x, y, t) = (0, 0, 0))$, $(\ell_1, (x, y, t) = (2, 2, 1))$, $(\ell_0, (x, y, t) = (3, 1, 2))$. □

In this work, we consider algorithms for linear invariant generation that work on LinTS's (as the abstract model). Informally, an invariant at a location is a logical formula that is always satisfied by the values of the variables whenever the location is entered by some path. An invariant is linear if it is a linear assertion. The formal definition is as follows.

(Linear) Invariants. An *invariant* at a location ℓ of a LinTS is a logical formula φ such that for every path under the LinTS $\pi = (\ell_0, \sigma_0) \dots (\ell_n, \sigma_n)$ and $0 \leq k \leq n$, it holds that $\ell_k = \ell$ implies $\sigma_k \models \varphi$. Furthermore, an invariant φ is *linear* if φ is a linear assertion over the variable set X.

To automatically generate invariants, one often investigates a strengthened notion called *inductive invariants*. Since we only consider linear invariants, we directly present the definition of inductive linear invariants, and in the form of inductive linear assertion maps.

(Inductive) Linear Assertion Maps. A *linear assertion map* over an LinTS is a function η that maps every location ℓ to a linear assertion $\eta(\ell)$ over the variables X. A linear assertion map η is *inductive* if the following conditions hold:

- (**Initialization**) $\theta \models \eta(\ell^*)$;
- (**Consecution**) For every transition $\tau = \langle \ell, \ell', \rho \rangle$, we have that $\eta(\ell) \wedge \rho \models \eta(\ell')'$, where $\eta(\ell')'$ is the linear assertion obtained by replacing every variable $x \in X$ in $\eta(\ell')$ with its next-value counterpart $x' \in X'$.

Informally, a linear assertion map is inductive if it is (i) implied by the initial condition given by θ at the initial location ℓ^* (i.e., **Initialization**) and (ii) preserved under the application of every transition (i.e., **Consecution**). By a straightfoward induction on the length of a path under a LinTS, one could verify that the linear assertion at every location in an inductive linear assertion map is guaranteed to be a linear invariant. In the rest of the work, we focus on the automated synthesis of inductive linear assertion maps.

3 AN OVERVIEW OF OUR APPROACH

In this section, we first review the original approaches in [18, 63] that generate linear invariants through Farkas' Lemma and point out the weakness of each approach, and then sketch our key improvements to these approaches.

3.1 The Original Approaches [18, 63]

In [18, 63], a sound and complete constraint-solving framework for linear invariant generation is proposed via Farkas' Lemma [30]. The use of Farkas' Lemma transforms the inductive condition for linear invariants equivalently into a system of quadratic constraints, by solving which one could obtain concrete linear invariants. The key merit of the use of Farkas' Lemma is that it simplifies the constraints from the inductive condition to quadratic constraints. In [18], the quadratic constraints obtained after applying Farkas' Lemma were solved exactly through quantifier elimination. While in [63], these constraints were solved instead through polyhedra manipulation by applying several reasonable heuristics to avoid the high runtime complexity caused by quantifier elimination.

To review these two approaches, we first recall Farkas' Lemma. Farkas' Lemma is a fundamental theorem that characterizes basic relationships between linear inequalities. Below we present the version of Farkas' Lemma that charactertizes the entailment from a linear assertion to a linear inequality. We follow the presentation form of Farkas' Lemma in [18].

Theorem 3.1 (Farkas' Lemma). *Consider a linear assertion φ over a set $V = \{x_1, \ldots, x_n\}$ of real-valued variables in the form of a conjunction of the following linear inequalities:*

$$\varphi : \begin{array}{c} a_{11} \cdot x_1 + \cdots + a_{1n} \cdot x_n + b_1 \geq 0 \\ \vdots \qquad\qquad \vdots \quad \vdots \\ a_{m1} \cdot x_1 + \cdots + a_{mn} \cdot x_n + b_m \geq 0 \end{array}$$

When φ is satisfiable (i.e., there is a valuation over V that satisfies φ), it implies a linear inequality ψ

$$\psi : c_1 \cdot x_1 + \cdots + c_n \cdot x_n + d \geq 0$$

(i.e., $\varphi \models \psi$) if and only if there exist non-negative real numbers $\lambda_0, \lambda_1, \ldots, \lambda_m$ such that (i) $c_j = \sum_{i=1}^{m} \lambda_i \cdot a_{ij}$ for all $1 \leq j \leq n$, and (ii) $d = \lambda_0 + \sum_{i=1}^{m} \lambda_i \cdot b_i$. Moreover, φ is unsatisfiable if and only if the inequality $-1 \geq 0$ (as ψ) can be derived from above.

Table 1. The Tabular Form for Farkas' Lemma

λ_0					1	$\geq$	0	
λ_1	$a_{11} \cdot x_1 + \cdots +$		$a_{1n} \cdot x_n +$	b_1	$\bowtie_1$	0	$\Big\}$	φ
$\vdots$	$\vdots$		$\vdots$	$\vdots$				
λ_m	$a_{m1} \cdot x_1 + \cdots +$		$a_{mn} \cdot x_n +$	b_m	$\bowtie_m$	0	$\Big\}$	
	$c_1 \cdot x_1 + \cdots +$		$c_n \cdot x_n +$	d	$\geq$	0	$\leftarrow$	ψ
				-1	$\geq$	0	$\leftarrow$	*false*

One direction of Farkas' Lemma is straightforward, as one easily sees that if we have a non-negative linear combination of the inequalities in φ that can derive ψ, then it is guaranteed that ψ holds whenever φ is true. Farkas' Lemma further establishes that the other direction is also valid.

Remark 1. *In the statement of Farkas' Lemma above, if we change a linear inequality $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j \geq 0$ in φ to equality (i.e., $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j = 0$), then the theorem holds with the relaxation that we do not require $\lambda_j \geq 0$. This could be easily observed by first replacing the equality equivalent with both $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j \geq 0$ and $a_{j1}x_1 + \cdots + a_{jn}x_n + b_j \leq 0$, and then applying Farkas' Lemma. By similar arguments, the theorem statement holds upon changing multiple linear inequalities into equalities with the relaxation of non-negativity for their corresponding λ_j's.*

The application of Farkas' Lemma can be visualized by the tabular form in Table 1, where $\bowtie_1, \ldots, \bowtie_m \in \{=, \geq\}$ and we mulitply $\lambda_0, \lambda_1, \ldots, \lambda_m$ with their inequalities in φ and sum up them together to get ψ. For $1 \leq j \leq m$, if $\bowtie_j$ is $\geq$, we require $\lambda_j \geq 0$, otherwise (i.e., $\bowtie_j$ is $=$) we do not impose restriction on λ_j. We then recall several concepts from polyhedra theory.

Polyhedra and polyhedral cones. A subset P of $\mathbb{R}^n$ is a *polyhedron* if $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}\}$ for some real matrix $A \in \mathbb{R}^{m \times n}$ and real vector $\mathbf{b} \in \mathbb{R}^m$, where $\mathbf{x}$ is treated as a column vector and the comparison $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ is defined in the coordinate-wise fashion. A polyhedron P is a *polyhedral cone* if $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A} \cdot \mathbf{x} \leq \mathbf{0}\}$ for some real matrix $A \in \mathbb{R}^{m \times n}$, where $\mathbf{0}$ is the m-dimensional zero column vector. It is well-known from Farkas-Minkowski-Weyl Theorem [64, Corollary 7.1a] that any polyhedral cone P can be represented as $P = \{\sum_{i=1}^{k} \lambda_i \cdot \mathbf{g}_i \mid \lambda_i \geq 0 \text{ for all } 1 \leq i \leq k\}$ for some real vectors $\mathbf{g}_1, \ldots, \mathbf{g}_k$, where such real vectors $\mathbf{g}_i$'s are called a collection of *generators* for the polyhedral cone P.

Polyhedron projection. For a polyhedron $P = \{(\mathbf{x}^{\mathrm{T}}, \mathbf{u}^{\mathrm{T}})^{\mathrm{T}} \in \mathbb{R}^{p+q} \mid \mathbf{A} \cdot \mathbf{x} + \mathbf{B} \cdot \mathbf{u} \leq \mathbf{c}\}$ where $\mathbf{A} \in \mathbb{R}^{m \times p}, \mathbf{B} \in \mathbb{R}^{m \times q}$ are real matrices and $c \in \mathbb{R}^m$ is a real vector, the projection of P onto the dimensions $\mathbf{x}$ (i.e., the first p dimensions) is defined as the polyhedron $P[\mathbf{x}] := \{\mathbf{x} \in \mathbb{R}^p \mid \exists \mathbf{u} \in \mathbb{R}^q.(\mathbf{x}^{\mathrm{T}}, \mathbf{u}^{\mathrm{T}})^{\mathrm{T}} \in P\}$. It is guaranteed by e.g. Fourier-Motzkin Elimination [64, Chapter 12.2] that $P[\mathbf{x}]$ will always be a polyhedron.

Now we review the approaches in [18, 63].

The invariant-generation workflow. Based on Farkas' Lemma, the approaches in [18, 63] generate linear invariants over a LinTS by the following steps (**Steps A1 – A3**). Below we fix an input LinTS with variables $X = \{x_1, \ldots, x_n\}$.

Step A1 In the first step, both the approaches establish a template for an inductive linear assertion maps. A template η involves a linear inequality $\eta(\ell) = c_{\ell,1} \cdot x_1 + \cdots + c_{\ell,n} \cdot x_n + d \geq 0$ at each location ℓ of the LinTS, such that the coefficients $c_{\ell,1}, \ldots, c_{\ell,n}, d$ are unknown and to be resolved.

Step A2 In the second step, both the approaches establish constraints from the initialization and the consecution conditions for invariants. The initialization condition specifies that the linear

inequality $\eta(\ell^*)$ at the initial location ℓ^* should be implied by the initial condition θ, i.e., $\theta \models \eta(\ell^*)$. The consecution condition specifies that every transition preserves the linear assertion map η, i.e., for every transition $\langle \ell, \ell', \rho \rangle$ we have that $\eta(\ell) \wedge \rho \models \eta(\ell')'$.

Step A3 In the third step, both the approaches apply Farkas' Lemma to the constraints collected from the initialization condition $\theta \models \eta(\ell^*)$ and the consecution conditions $\eta(\ell) \wedge \rho \models \eta(\ell')'$ for each transition $\langle \ell, \ell', \rho \rangle$. For intialization, we apply the tabular form (Table 1) to obtain

$$
\begin{array}{c|l}
\lambda_0 & 1 \;\geq\; 0 \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + b_1 \;\bowtie_1\; 0 \\
\vdots & \vdots \qquad\qquad \vdots \quad \vdots \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + b_m \;\bowtie_m\; 0 \\
\hline
 & c_{\ell^*,1}x_1 + \cdots + c_{\ell^*,n}x_n + d_{\ell^*} \;\geq\; 0 \;\leftarrow\; \eta(\ell^*) \\
 & -1 \;\geq\; 0 \;\leftarrow\; \mathit{false}
\end{array}
\quad \Big\} \; \theta
$$

which results in a linear assertion over the unknown coefficients $c_{\ell^*,1}, \ldots, c_{\ell^*,n}, d$ and the fresh variables $\lambda_0, \lambda_1, \ldots, \lambda_m$. Similarly, the tabular form for the consecution condition of a transition $\langle \ell, \ell', \rho \rangle$ gives

$$
\begin{array}{c|l}
\mu & c_{\ell,1}x_1 + \cdots + c_{\ell,n}x_n + d_\ell \;\geq\; 0 \leftarrow\; \eta(\ell) \\
\lambda_0 & 1 \;\geq\; 0 \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + a'_{11}x'_1 + \cdots + a'_{1n}x'_n + b_1 \;\bowtie_1\; 0 \\
\vdots & \vdots \qquad\qquad \vdots \qquad\qquad \vdots \quad \vdots \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n + b_m \;\bowtie_m\; 0 \\
\hline
 & c_{\ell',1}x'_1 + \cdots + c_{\ell',n}x'_n + d_{\ell'} \;\geq\; 0 \leftarrow \eta(\ell')' \\
 & -1 \;\geq\; 0 \leftarrow\; \mathit{false}
\end{array}
\quad \Big\} \; \rho
$$

where in addition to $\lambda_j, c_{\ell,j}, d_\ell, c_{\ell',j}, d_{\ell'}$ we have a fresh variable μ as the non-negative muliplier for $\eta(\ell)$. Note that for the consecution condition, the constraint obtained is no longer linear since the fresh variable μ is multiplied to $\eta(\ell)$ in the tabular above.

Step A4 In the last step, the (non-linear) constraints collected in the previous step are solved to obtain the concrete values for the unknown coefficients in η, so that a concrete inductive linear assertion map would be obtained. In [18], these constraints were solved through quantifier elimination, while in [63] the constraints were solved through (i) several reasonable heuristics to guess possible values for the key parameter μ so as to remove the non-linearity and (ii) the generator computation of a polyhedral cone originally represented in its linear inequalities. A major heuristics adopted in [63] to guess possible values for μ is based on some practical rules such as factorization and setting μ manually to $0, 1$ (where 0 means an invariant local to the guard of the transition and 1 means an invariant incremental to the previous step).

Below we focus on the workflow of [63], since it is the most related work to our result. Since the approach of [63] resolves the mulipliers μ for consecution by guessing their concrete values, the guessed values for the consecution condition of every transition in a LinTS result in a disjunction of linear assertions over the unknown coefficients in the template η, where each guessed value for the parameter μ corresponds to one linear assertion in the disjunction. By combining conjunctively the constraints obtained from the consecution condition (for every tranisition) and those from the initial condition, we obtain a propositional formula in conjunctive normal form (CNF) where each atomic proposition is a linear assertion over the unknown coefficients of η. More specifically, each atomic proposition is in the form $\mathbf{A} \cdot \mathbf{c} \leq \mathbf{0}$ where $\mathbf{c}$ is the vector of unknown coefficients in the template, hence is a polyhedral cone. Having obtained the CNF formula, the approach expands the formula (through the distributive law between conjunction and disjunction) equivalently into a DNF (disjunctive normal form) formula where each disjunctive clause is a conjunction of certain

atomic propositions in the original CNF formula (therefore being still a polyhedral cone), and then computes through the double description method [8] and Parma Polyhedra library (PPL) [8, 58] the generators of each disjunctive clause (in the form of a polyhedral cone) once a disjunctive clause is expanded. By instantiating the computed generators back to the unknown coefficients in the template, the approach obtains the solved concrete linear invariants, for which one generator corresponds to one inductive linear asertion map.

A key improvement to reduce the runtime arising from the expansion from CNF to DNF in [63] is the *subsumption testing* that checks whether the current conjunction of atomic propositions during the expansion is already subsumed by (i.e., implies) the invariants obtained from the previously-expanded disjunctive clauses; once the subsumption is successful, one knows that the current conjunction will not produce more meaningful invariants, hence the computation for the current conjunction halts and the expansion proceeds by switching to other branches.

Below we present an example to illustrate the workflow of [63].

Example 2. *Consider the LinTS in Example 1. The approach in [63] first establishes a template η by setting $\eta(\ell_i) := c_{\ell_i,1}x + c_{\ell_i,2}y + c_{\ell_i,3}t + d_{\ell_i} \geq 0$ for $i \in \{0, 1\}$. Then the approach encodes initialization and consecution by the tabular form in Table 1. For the initialization, we have*

$$\begin{array}{c|cccc|l} \lambda_0 & & & & 1 & \geq 0 & \\ \lambda_1 & x & & & & = 0 & \Big\} \\ \lambda_2 & & y & & & = 0 & \Big\} \ \theta \\ \lambda_3 & & & t & & = 0 & \Big\} \\ \hline & c_{\ell_0,1}x + c_{\ell_0,2}y + c_{\ell_0,3}t + d_{\ell_0} \geq 0 & & & & & \leftarrow \eta(\ell_0) \end{array}$$

that results in the constraints $\left[c_{\ell_0,1} = \lambda_1, c_{\ell_0,2} = \lambda_2, c_{\ell_0,3} = \lambda_3, d_{\ell_0} \geq 0\right]$. After projecting away the fresh variables λ_j's, we obtain $[d_{\ell_0} \geq 0]$ for the initialization. For the consecution conditions, we present the tabular form for the transition τ_1:

$$\begin{array}{c|cccccccc|l} \mu & c_{\ell_0,1}x + c_{\ell_0,2}y + c_{\ell_0,3}t & & & & & & + d_{\ell_0} & \geq 0 & \leftarrow \eta(\ell_0) \\ \lambda_0 & & & & & & & 1 & \geq 0 & \\ \lambda_1 & -x & + & t & + & x' & - & t' & \geq 0 & \Big\} \\ \lambda_2 & x & - & 2t & - & x' & + & 2t' & \geq 0 & \Big\} \\ \lambda_3 & & -y & - & 2t & + & y' & + 2t' & \geq 0 & \Big\} \\ \lambda_4 & & y & + & t & - & y' & - & t' & \geq 0 & \Big\} \ \rho_1 \\ \lambda_5 & & & - & t & & & & t' - 1 & \geq 0 & \Big\} \\ \lambda_6 & & & & t & & & - & t' + 2 & \geq 0 & \Big\} \\ \hline & & & & & & c_{\ell_1,1}x' + c_{\ell_1,2}y' + c_{\ell_1,3}t' + d_{\ell_1} & & & \geq 0 & \leftarrow \eta(\ell_1)' \end{array}$$

where the fresh variables λ_j's are local to the tabular above and does not overlap with the tabulars for other transitions. The λ_j's are again eliminated by polyhedron projection, while the fresh variable μ is eliminated by heuristically guessing its values. The transition τ_2 is treated in the similar way.

After guessing the values for μ with several heuristics, the resultant constraint is a CNF formula where each atomic proposition is a polyhedral cone over the unknown coefficients. The approach further expands the CNF equivalently into a DNF formula (which we omit here due to its vast amount of technicality). For this example, one disjunctive clause from the DNF formula is as follows (where we abbreviate $c_{\ell_i,j}$ as c_{ij}, d_{ℓ_i} as d_i):

$$\begin{bmatrix} c_{01}=c_{11}, & c_{02}=c_{12}, & c_{03}=c_{13}, & d_0\geq 0, \\ 2c_{01}-d_0+c_{12}+c_{13}+d_1\geq 0, & 2c_{01}-d_0+2c_{12}+c_{13}+d_1\geq 0, \\ 4c_{01}-d_0+4c_{12}+2c_{13}+d_1\geq 0, & c_{01}-d_0+c_{12}+c_{13}+d_1\geq 0, \\ c_{01}-d_0+2c_{12}+c_{13}+d_1\geq 0, & 2c_{01}-d_0+4c_{12}+2c_{13}+d_1\geq 0, \\ 2c_{01}+d_0-4c_{12}+2c_{13}-d_1\geq 0, & 4c_{01}+d_0-4c_{12}+2c_{13}-d_1\geq 0, \\ c_{01}+d_0-2c_{12}+c_{13}-d_1\geq 0, & 2c_{01}+d_0-2c_{12}+c_{13}-d_1\geq 0, \\ c_{01}+d_0-c_{12}+c_{13}-d_1\geq 0, & 2c_{01}+d_0-c_{12}+c_{13}-d_1\geq 0 \end{bmatrix} \tag{1}$$

By computing the generators of the polyhedral cone above, we obtain the following generators and their corresponding invariants in Table 2, where in the left part each row specifies a generator (over the unknown coefficients c_{ij}, d_i's) and in the right part we instantiate the generator to the unknown coefficients in the template η to obtain the invariants at locations ℓ_0 and ℓ_1.

Table 2. Generators (left) and Their Invariants (right) for (1)

c_{01}	c_{02}	c_{03}	d_0	c_{11}	c_{12}	c_{13}	d_1	$\eta(\ell_0)$	$\eta(\ell_1)$
0	0	0	1	0	0	0	1	$1\geq 0$	$1\geq 0$
0	0	0	0	0	0	0	0	$0\geq 0$	$0\geq 0$
−1	0	2	0	−1	0	2	0	$-x+2t\geq 0$	$-x+2t\geq 0$
0	−1	1	0	0	−1	1	2	$-y+t\geq 0$	$-y+t+2\geq 0$
0	−1	2	0	0	−1	2	0	$-y+2t\geq 0$	$-y+2t\geq 0$
0	0	1	0	0	0	1	1	$t\geq 0$	$t+1\geq 0$
0	1	2	0	0	1	2	0	$y+2t\geq 0$	$y+2t\geq 0$
0	1	1	0	0	1	1	−2	$y+t\geq 0$	$y+t-2\geq 0$
1	0	−1	0	1	0	−1	0	$x-t\geq 0$	$x-t\geq 0$
0	0	1	0	0	0	1	−1	$t\geq 0$	$t-1\geq 0$

The obtained invariants are further minimized, so that the final invariants obtained at $\eta(\ell_0)$ are

$$\begin{bmatrix} -x+2t\geq 0 & -y+t\geq 0 & x-t\geq 0 & y+t\geq 0 \end{bmatrix}$$

and for $\eta(\ell_1)$ the invariants are

$$\begin{bmatrix} -x+2t\geq 0 & -y+t+2\geq 0 & -y+2t\geq 0 \\ t-1\geq 0 & y+t-2\geq 0 & x-t\geq 0 \end{bmatrix}.$$

It happens that the invariants from the whole DNF formula coincide with those computed from (1). □

Weakness of [18, 63]. Although the approaches [18, 63] provides the first and only invariant generation approaches via Farkas' Lemma, they have the following weakness. The approach [18] generates the invariants by quantifier elimination, thus is impractical [78]. The approach [63] avoids the high runtime complexity from quantifier elimination by several reasonable heuristics in the application of Farkas' Lemma, solves the invariants by generator computation, and performs subsumption testing to reduce the combinatorial explosion from the CNF-to-DNF expansion; however, the CNF-to-DNF expansion with subsumption testing still leads to a large amount of combinatorial explosion, and the generator computation also causes a considerable amount of combinatorial explosion.

In order to further reduce combinatorial explosion in [63], we complement the approach [63] with (i) a location-by-location idea that generates the invariants one program location at a time and (ii) a segmented subsumption testing that is independent of the original subsumption testing in [63]. The following subsections illustrate the main idea of our improvements.

3.2 Location-by-Location Linear Invariant Generation

A drawback of the approach [63] is that in its last step (**Step A4** in the previous subsection), the invariants are obtained as the generators of a whole polyhedral cone that involves the unknown coefficients at all locations. This leads to two obstacles that may largely increase the runtime. The first is that in the subsumption testing, there is only a small chance that a whole polyhedral cone with all the unknown coefficients be subsumed by the invariants already generated. The second is that the generator computation of a polyhedral cone is an expensive operation since it may cause exponential blowup in the number of its variables, thus to compute the generators of a whole polyhedral cone that involves all the unknown coefficients may induce a large amount of runtime.

To address the two obstacles, we propose the novel idea of generating the invariants one location at a time (i.e., location-by-location). With this idea, the chance of successful subsumption in [63] is increased since now the subsumption testing involves only one location. Moreover, this idea directly leads to a parallel processing that breaks down the whole invariant generation task at all locations into multiple processors for which each processor only handles a small number of program locations. Note that although the idea seems simple, it can directly lead to improvement on the runtime, and enables further technical improvements.

Below we present an example to illustrate our idea.

Example 3. *Consider Example 2 again. We run **Steps A1–A3** of the approach [63] and the part of **Step A4** that computes a CNF from the previous steps. Recall that originally in **Step A4**, the approach [63] expands the CNF into its equivalent DNF, and computes the generators for each atomic proposition (polyhedral cone) of the DNF. Our location-by-location idea distinguish itself by focusing on one target location in the whole invariant generation process and generate only the invariants at the target location accordingly, and a simple way to implement this idea is to project the polyhedral cones in the DNF onto the unknown coefficients related to the target location. For example, suppose that we focus on the target location ℓ_0. Then a simple way to implement our idea is to project the polyhedral cones in the DNF onto the unknown coefficients related to ℓ_0. For instance, for the polyhedral cone in 1, we project the polyhedral cone onto the dimensions specified by the unknown coefficients c_{0j}'s $(1 \leq j \leq 3)$ and d_0 that are related to the location ℓ_0, to obtain the following polyhedral cone*

$$\begin{bmatrix} d_0 \geq 0 & c_{01} - c_{02} + c_{03} \geq 0 & c_{01} + c_{02} + c_{03} \geq 0 \\ & 2c_{01} - c_{02} + c_{03} \geq 0 & 2c_{01} + c_{02} + c_{03} \geq 0 \end{bmatrix}.$$

The generators of this projected polyhedral cone gives the following linear invariants at the location ℓ_0:

c_{01}	c_{02}	c_{03}	d_0	$\eta(\ell_0)$
0	0	0	1	$1 \geq 0$
0	0	0	0	$0 \geq 0$
1	0	−1	0	$x - t \geq 0$
0	1	1	0	$y + t \geq 0$
−1	0	2	0	$-x + 2t \geq 0$
0	−1	1	0	$-y + t \geq 0$

After minimization, the invariants added for the location ℓ_0 from the polyhedral cone (1) include

$$\begin{bmatrix} -x + 2t \geq 0 & -y + t \geq 0 & x - t \geq 0 & y + t \geq 0 \end{bmatrix}.$$

which happen to include all the linear inductive invariants at ℓ_0. Note that under our idea, we only generate the invariants at ℓ_0 in one invariant-generation process, and keep the invariants at other

locations (i.e., ℓ_1) to be ***true****. By another invariant-generation process for the location ℓ_1, we get the invariants at the location ℓ_1 as follows:*

$$\begin{bmatrix} -x+2t \geq 0 & -y+t+2 \geq 0 & -y+2t \geq 0 \\ t-1 \geq 0 & y+t-2 \geq 0 & x-t \geq 0 \end{bmatrix}.$$

The invariants obtained coincide with the approach in [63]. □

3.3 Segmented Subsumption Testing

We also propose a novel subsumption testing on the CNF-to-DNF expansion in **Step A4** of Section 3.1. The main idea is that the subsumption testing divides the CNF formula into several segments and perform local subsumption testing in each segment. Our subsumption testing is independent of the original one in [63] since we focus on local subsumption in each segment, while the original one [63] focuses however on global subsumption testing on the whole CNF formula. Thus, our subsumption testing is able to find local subsumptions in advance so that the original CNF formula can be simplified before it is used to generate invariants.

Below we present a scenario to illustrate this idea.

Example 4. *Consider the scenario that the CNF formula to be expanded in* ***Step A4*** *is $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ where each C_i is a disjunction of polyhedral cones. The idea of segmented subsumption divides the CNF into two segments $C_1 \wedge C_2$ and $C_3 \wedge C_4$, and perform local subsumption testing in each segment. For the segment $C_1 \wedge C_2$ (as a sub-CNF formula), we expand it into a (sub-)DNF and remove redundant disjunctive clauses by subsumption testing, to obtain a DNF formula Θ_1. We also perform the same operation to $C_3 \wedge C_4$ and obtain a DNF Θ_2. Finally, we apply our improvements in the previous subsection to the CNF $\Theta_1 \wedge \Theta_2$ (which is equivalent to Φ) to generate the invariants.* □

4 LOCATION-BY-LOCATION LINEAR INVARIANT GENERATION

In this section, we formally demonstrate our idea of generating the linear invariants at only one target program location in each invariant generation process, thus generating the linear invariants location-by-location. The idea could be easily implemented by projecting the generated invariants onto the (related coefficents at) the target location in **Step A4** of Section 3.1, and leads directly to a parallel processing that assigns the invariant generation task at every program location separately onto different processors. Based on the idea, in the following we present two technical improvements that further reinforce the idea by reducing more combinatorial explosion in the original approach [63]. The first improvement is a reordering of the expansion from the CNF into its equivalent DNF at **Step A4** (Section 3.1) so that the transitions associated with the target program location are expanded first to detect possible subsumptions earlier. The second improvement is a technique to compute the generators (as the generated invariants, see **Step A4** of Section 3.1) at the target location via the projection onto the location. Below we fix a LinTS and a target location $\ell_\star$ on which invariant generation is considered.

4.1 Reordered CNF-to-DNF Expansion

Recall that in **Step A4**, we obtain a CNF formula and expand the CNF into its equivalent DNF from which we generate invariants. Since the transformation from CNF into its DNF causes combinatorial explosion, the original approach [63] adopts a subsumption testing that detects redundant disjunctive clauses in the DNF. Here, since we focus only on the target location $\ell_\star$, we can reorder the expansion for the CNF formula by expanding the transitions mostly related to the target location first so that subsumptions could be detected much earlier than the original approach. The technical details is as follows.

Reordering of the CNF-to-DNF Expansion. We reorder the expansion from the CNF to its DNF so that the transitions of the LinTS associated with $\ell_\star$ are expanded first, and the other transitions are expanded last. The reason to expand the transitions with the target location $\ell_\star$ first is as follows: since we focus on the invariant generation at $\ell_\star$, successful subsumptions would be detected earlier if we check transitions that involve the target location $\ell_\star$ first; in contrast, if one chooses an arbitary order for the expansion, then from our idea the expansion at locations other than $\ell_\star$ cannot lead to successful subsumptions, as the subsumption testing is operated by checking the polyhedral inclusion that only involves the unknown coefficients at $\ell_\star$ (i.e., not involving the unknown coefficients at other locations). We further refine the epansion order by having that the *intra*-transitions in the LinTS that involves only the location $\ell_\star$ (i.e., the transitions from $\ell_\star$ to itself) are expanded first, those that involves $\ell_\star$ and other locations (i.e., the *inter*-transitions from $\ell_\star$ to other locations or from other locations to $\ell_\star$) are expanded second, those that does not involve location $\ell_\star$ are expanded last; the heuristics behind the refined reordering is that they are ordered by the descending order of relevance to the target location $\ell_\star$.

The pseudo-code for the invariant-generation algorithm that adopts the location-by-location idea and integrates the reordered expansion is given in Algorithm 1. In the pseudo-code, the algorithm first reorders the CNF formula $\bigwedge_i C_i$ obtained from **Step A4** in Section 3.1 as stated in the previous paragraph (line 1). The reordered CNF is $\bigwedge_i C'_i$. Then the algorithm initializes the values for $inv(\ell_\star)$, i and all n_i's (lines 2–4): the variable $inv(\ell_\star)$ is a set variable used to collect linear invariants generated in the algorithm and initialized to be $\emptyset$; since we implement expansion from the reordered CNF to its DNF through backtracking, we use the variable i to represent that the algorithm is currently traversing the atomic propositions in C'_i, and the variable n_i to represent that the current atomic proposition being traversed in C'_i is the n_i-th atomic proposition. Next, the algorithm follows the backtracking process that iteratively selects one atomic proposition d_i from each C'_i (line 7) following the increasing order of i, aggressively checks whether the current exploration has already been subsumed by the current invariant $inv(\ell_\star)$ (line 9), and updates the current invariant set by adding the invariants that can be derived from the conjunction $\bigwedge_{s=1}^{m} d_s$ (as a polyhedral cone) and *only* involve the unknown coefficients at the target location $\ell_\star$, which is done through the **Gen** procedure if all the preceding subsumption testings have been unsuccessful. Note that in the subsumption testing, the condition $\bigwedge_{s=1}^{i} d_s \not\subseteq inv(\ell_\star)$ means that the polyhedral cone defined by the linear assertion $\bigwedge_{s=1}^{i} d_s$ is not a subset of the counterpart generated by the linear invariants (as generators extended to full dimension) in $inv(\ell_\star)$. The return value is the final set value of $inv(\ell_\star)$ after the whole backtracking process, which collects all the linear invariants that are generators of the polyhedral cones having passed the subsumption testing.

A missing part in the pseudo-code is the details for the procedure **Gen**, whose counterpart in the original work [63] is the generator computation of the polyhedral cone that involves the unknown coefficients at all program locations. In the next part, we instantiate the procedure by our second improvement related to location-by-location linear invariant generation that proposes various procedures to compute the generators that only involve the dimensions at the target location.

4.2 Projection-Based Generator Computation

In the original work [63], the final invariants are generated by computing the generators of the polyhedral cones (i.e., *poly* at line 11 in Algorithm 1) obtained from the transformed DNF. As stated previously, the polyhedral cones involve the unknown coefficients at all program locations, thus the generation computation over these polyhedral cones may cause combinatorial explosion. From our idea of generating the invariants only at the target location, one only needs to compute the generators that involve the dimensions at the target location. This allows further technical

Algorithm 1 Linear Invariant Generation at Location $\ell_\star$

Input: $\ell_\star$: target location;
$\bigwedge_{i=1}^{m} C_i$: the CNF formula obtained from **Step A4** (Section 3.1) in the approach [18] where each C_i is the i-th conjunctive clause and is a disjunction of polyhedra over the unknown coefficients in the template;
Reordering$(\bigwedge_i C_i)$: a procedure that reorders all C_i's into $\bigwedge_i C'_i$ w.r.t whether each C_i corresponds to an intra- or an inter-transition as stated previously; we write $C'_i = \bigvee_{j=1}^{N_i} c'_{i,j}$ where each $c'_{i,j}$ is an atomic proposition in the form of a polyhedron over the unknown coefficients;
Gen$(\ell, poly)$: a procedure that generates the invariants at a location ℓ given a polyhedron $poly$ over the unknown coefficients
Output: a collection $inv(\ell_\star)$ of linear invariants at $\ell_\star$;

$\bigwedge_{i=1}^{m} C'_i \leftarrow$ **Reordering**$(\bigwedge_{i=1}^{m} C_i)$; //$C'_i = \bigvee_{j=1}^{N_i} c'_{i,j}$
$inv(\ell_\star) \leftarrow \emptyset$;
$i \leftarrow 1$;
$n_i \leftarrow 1$ (for $1 \le i \le m$);
while $i > 0$ **do**
  if $n_i \le N_i$ **then**
    $d_i \leftarrow c'_{i,n_i}$; //select one atomic proposition
    $n_i \leftarrow n_i + 1$;
    if $\bigwedge_{s=1}^{i} d_s \not\subseteq inv(\ell_\star)$ **then** //subsumption
      if $i = m$ **then**
        $inv(\ell_\star) \leftarrow inv(\ell_\star) \cup$ **Gen**$(\ell_\star, \bigwedge_{s=1}^{m} d_s)$;
      else
        $i \leftarrow i + 1$;
      end if
    end if
  else
    $n_i \leftarrow 1$; //backtracking
    $i \leftarrow i - 1$;
  end if
end while;
return $inv(\ell_\star)$;

improvements that can reduce the combinatorial explosion from the generator computation. Below we propose several solutions that compute generators only at the dimensions of the target location. These solutions are to be substituted into the procedure **Gen** in the pseudo-code of Algorithm 1 to complete our invariant-generation procedure for location-by-location linear invariant generation.

Generator Computation at Target Location We propose four solutions to compute the invariants at the dimensions of the target location, namely generator projection, Fourier-Motzkin Elimination (FME), Block Elimination and our improvement on Block Elimination. Below we fix a polyhedral cone $P = \{(\mathbf{x}^{\mathrm{T}}, \mathbf{u}^{\mathrm{T}})^{\mathrm{T}} \in \mathbb{R}^{p+q} \mid \mathbf{Ax} + \mathbf{Bu} \le \mathbf{c}\}$ and consider to compute the generators only over the dimensions $\mathbf{x}$ that correspond to the dimensions of unknown coefficients at our target location $\ell_\star$.

Generator Projection. Our first solution is still to compute the generators directly for the polyhedral cones that involves all program locations by standard methods (e.g., the double-description method [33]), but to project the computed generators onto the unknown coefficients at the target

location $\ell_\star$. Thus in detail, the first solution computes the generators of the polyhedral cone P and project the generators to the dimensions $\mathbf{x}$.

Fourier-Motzkin Elimination. Our second solution is to apply the classical method of Fourier-Motzkin Elimination (FME) [64, Chapter 12.2] that first projects away the extra dimensions other than the unknown coefficients at the target location $\ell_\star$ and then computes the generators. In detail, the solution through FME first eliminates the variables $\mathbf{u}$ one by one in P to get the projected polyhedral cone $P[\mathbf{x}]$ on the dimensions $\mathbf{x}$, then compute the generators of $P[\mathbf{x}]$ by standard methods.

Block Elimination. We consider applying Block Elimination (see e.g. [73]) as our third solution, which is an alternative approach that eliminate unrelated variables through the computation of the generators of the *projection cone* of a polyhedron. The motivation is to address the issue of generating possibly many redundant inequalities in the application of FME. The details of Block Elimination is as follows. To illustrate this method, we first present a variant form of Farkas' Lemma [64, Corollary 7.1e].

Theorem 4.1. *Let* $\mathbf{A}$ *be a real matrix and* $\mathbf{b}$ *be a real vector. The polyhedron* $P = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\}$ *is non-empty if and only if there is no vector* $\mathbf{y}$ *satisfying* $\mathbf{y} \ge \mathbf{0}$, $\mathbf{y}^{\mathrm{T}}\mathbf{A} = \mathbf{0}$ *and* $\mathbf{y}^{\mathrm{T}}\mathbf{b} < 0$.

The polyhedral cone $\{\mathbf{y} \mid \mathbf{y} \ge \mathbf{0} \wedge \mathbf{y}^{\mathrm{T}}\mathbf{A} = \mathbf{0}\}$ derived from the polyhedron $P = \{\mathbf{x} \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\}$ in the statement of Theorem 4.1 is usually called the *projection cone* of P, which we denoted by $\mathit{proj}(\mathbf{A})$ since it is only related to the matrix $\mathbf{A}$ (that help defines P).

Block Elimination utilizes projection cones to reduce the amount of redundant inequalities that may arise in FME. Consider the polyhedron $P = \{(\mathbf{x}^{\mathrm{T}}, \mathbf{u}^{\mathrm{T}})^{\mathrm{T}} \in \mathbb{R}^{p+q} \mid \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \le \mathbf{c}\}$ and the projection of P onto $\mathbf{x}$. The method treats P as a polyhedron $Q(\mathbf{x})$ with the parameter $\mathbf{x}$ such that $Q(\mathbf{x}) := \{\mathbf{u} \in \mathbb{R}^q \mid \mathbf{B}\mathbf{u} \le \mathbf{c} - \mathbf{A}\mathbf{x}\}$ and computes the generators of the projection cone $\mathit{proj}(\mathbf{B})$. By Theorem 4.1, to ensure that $Q(\mathbf{x})$ is non-empty, it suffices to guarantee that $\mathbf{g}^{\mathrm{T}}(\mathbf{c} - \mathbf{A}\mathbf{x}) \ge 0$ for all the generators $\mathbf{g}$ of $\mathit{proj}(Q(\mathbf{x}))$. Hence, after the generators $\mathbf{g}_1, \ldots, \mathbf{g}_k$ of the projection cone $\mathit{proj}(Q(\mathbf{x}))$ are computed, the method outputs the projected polyhedron $P[\mathbf{x}]$ as defined by the linear inequalities $\mathbf{g}_j^{\mathrm{T}}(\mathbf{c} - \mathbf{A}\mathbf{x}) \ge 0$ for $1 \le j \le k$. The remaining task is again to compute the generators of the projected polyhedral cone $P[\mathbf{x}]$.

Our improvement on Block Elimination. Recall that our task is to compute the projected generators of a polyhedral cone $\mathbf{A}\mathbf{c} \le \mathbf{0}$ onto the unknown coefficients at $\ell_\star$ (here $\mathbf{c}$ is the vector of all unknown coefficients in the template), where the polyhedral cone $\mathbf{A}\mathbf{c} \le \mathbf{0}$ is a disjunctive clause from the DNF obtained in **Step A4**. Following Block Elimination, we still project the polyhedral cone $\mathbf{A}\mathbf{c} \le \mathbf{0}$ onto the unknown coefficients at $\ell_\star$, but have further improvements as follows.

Below we describe our improvements in detail. Let $\mathbf{c}_\star$ be the vector of unknown coefficients at $\ell_\star$ and $\mathbf{c}_*$ be the vector of unknown coefficients at other locations. Then $\mathbf{A}\mathbf{c} \le \mathbf{0}$ can be decomposed as

$$\mathbf{A}\mathbf{c} = \begin{pmatrix} \mathbf{F} & \mathbf{0} \\ \mathbf{G} & \mathbf{G}' \\ \mathbf{0} & \mathbf{H} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{c}_* \\ \mathbf{c}_\star \end{pmatrix} \le \mathbf{0}$$

where $\mathbf{F}\mathbf{c}_* \le \mathbf{0}$ include the inequalities that involve only variables in $\mathbf{c}_*$, $\mathbf{G}\mathbf{c}_* + \mathbf{G}'\mathbf{c}_\star \le \mathbf{0}$ include the inequalities that involve both $\mathbf{c}_*$ and $\mathbf{c}_\star$, and $\mathbf{H}\mathbf{c}_\star \le \mathbf{0}$ include the inequalities that involve only variables in $\mathbf{c}_\star$. Since $\mathbf{H}\mathbf{c}_\star \le \mathbf{0}$ is not related to $\mathbf{c}_*$, our first (slight) improvement is to consider only

$$\mathbf{B}\mathbf{c}_* \le \mathbf{b} \text{ with } \mathbf{B} := \begin{pmatrix} \mathbf{F} \\ \mathbf{G} \end{pmatrix} \text{ and } \mathbf{b} := \begin{pmatrix} \mathbf{0} \\ -\mathbf{G}' \cdot \mathbf{c}_\star \end{pmatrix} \tag{2}$$

in the projection onto $\mathbf{c}_\star$. Furthermore, in the projection cone $\mathit{proj}(\mathbf{B})$:

$$\begin{pmatrix} \mathbf{y}_* \\ \mathbf{y}_\star \end{pmatrix} \ge \mathbf{0},\ (\mathbf{y}_*^{\mathrm{T}},\ \mathbf{y}_\star^{\mathrm{T}}) \cdot \begin{pmatrix} \mathbf{F} \\ \mathbf{G} \end{pmatrix} = \mathbf{0}, \tag{3}$$

we only need to consider the generators for $(\mathbf{y}_*^{\mathrm{T}}, \mathbf{y}_\star^{\mathrm{T}})^{\mathrm{T}}$ projected onto the dimensions $\mathbf{y}_\star$ that correspond to the rows of $\mathbf{G}$ since the vector $\mathbf{b}$ in (2) has all zero at the dimensions $\mathbf{y}_*$. Thus, our second (major) improvement is to project the projection cone $\mathit{proj}(\mathbf{B})$ onto $\mathbf{y}_\star$, and then compute the generators of the projected cone on the dimensions $\mathbf{y}_\star$ instead of on the full dimension $(\mathbf{y}_*^{\mathrm{T}}, \mathbf{y}_\star^{\mathrm{T}})^{\mathrm{T}}$.

4.3 Correctness and Accuracy

We show that the improvements through our location-by-location idea have the same accuracy as compared with the original work [63], thus generate correct linear invariants. This can be easily observed as follows. Suppose that the CNF formula obtained from **Step A4** is $\Phi = \bigwedge_i \bigvee_{j=1}^{N_i} c_{i,j}$ and we denote by $[\![\Phi]\!]$ the set of all real vectors (as valuations over all unknown coefficients in the template) that satisfy Φ. The approach [63] computes a set $\mathcal{G}$ of generators by first expanding the CNF Φ into DNF and then collecting the generators of every unsubsumped disjunctive clause of the DNF. Hence, from [63] we have that the convex cone generated by $\mathcal{G}$ is equal to that of $[\![\Phi]\!]$. In our improvements, we have that (i) the expansion reordering does not lose accuracy since the reordered CNF Φ' is logically equivalent to Φ, and (ii) the projection based generator computation preserves accuracy since always the same generators will be produced by either (i) first computing the generators then performing the projection onto the target location or (ii) first performing the projection then computing the generators. Hence the convex cone generated by the set of generators computed by our improvements is equal to that by $[\![\Phi]\!]$ projected to the target location.

5 SEGMENTED SUBSUMPTION TESTING

In this part, we propose an independent improvement on the CNF-to-DNF expansion in **Step A4** of Section 3.1. Recall that in Section 4.1, we have already proposed an improvement on the CNF-to-DNF by reordering of the expansion order to detect possible subsumptions earlier. The improvement proposed in this part, called segmented subsumption, follows a different idea: we consider to detect local subsumptions in a segment of the whole CNF, so that such local subsumptions can be detected in advance before taking the improvements described in Section 4.

To illustrate our improvement, we depict the general CNF-to-DNF expansion in Figure 2. In the figure, we represent the original CNF formula as $\bigwedge_{i=1}^{m} C_i$ where we have that each $C_i = \bigvee_{j=1}^{N_i} c_{i,j}$, with each $c_{i,j}$ being a polyhedral cone; we render the conjunctive clauses $C_1, \ldots, C_m$ in the row-wise fashion, for which the vertical order from C_1 to C_m is the order in the CNF-to-DNF expansion and the arrows represent the directions of the expansion in one step. We also define the notion of paths based on Figure 2. A *path* π in Figure 2 is of the form

$$c_{p,k_p} \longrightarrow \cdots \longrightarrow c_{i,k_i} \longrightarrow \cdots \longrightarrow c_{q,k_q}$$

where $1 \le p \le q \le m$ and $1 \le k_i \le N_i$ for all $p \le i \le q$. In other words, a path reflects a partial expansion from C_p to C_q where the literals chosen in the expansion are $c_{p,k_p}, \ldots, c_{i,k_i}, \ldots, c_{q,k_q}$. Note that if $p = 1$ and $q = m$, then we call the path π *complete* and it corresponds to a disjunctive clause $\wedge_{i=p}^{q} c_{i,k_i}$ in the transformed DNF, otherwise it is *incomplete*.

Below we describe the main idea of our segmented subsumption by an illustrative example.

Example 5. *Consider a specific scenario of Figure 2 that $m = 4$ and $k_i = 2$ for all $1 \le i \le 4$. This scenario is depicted in Figure 3. In this scenario, we consider the situation that the complete path $\pi = c_{1,1} \longrightarrow c_{2,1} \longrightarrow c_{3,1} \longrightarrow c_{4,1}$ subsumes all other complete paths in the expansion, i.e., $\bigwedge_i c_{i,k_i} \subseteq \bigwedge_i c_{i,1}$ for all the choices of k_i's. In the original subsumption testing from [63], the other complete paths are expanded and are checked subsumption with the first complete path π in the expansion; thus, there could be 15 subsumptions detected by the subsumption testing. Note the reordering in Section 4.1 is irrelevant here as we can assume that the order C_1 to C_4 in Figure 3 is already reordered. Although the original*

approach in [63] successfully detected all the subsumptions, it would still perform 15 *subsumption testings, which would be time-consuming when there is a large amount of such subsumptions.*

In order to further reduce the number of subsumption testing as shown above, we consider to detect local subsumptions in advance. In detail, for the scenario of Figure 3, our local subsumption testing is as follows. We first divide C_1, C_2, C_3, C_4 *into two segmments,* $C_1 \wedge C_2$ *and* $C_3 \wedge C_4$*. Then in each segment, we perform local subsumption testing. For example, for the segment* $\{C_1, C_2\}$*, the local subsumption testing first expand* $c_{1,1} \wedge c_{2,1}$ *and then checks whether* $\bigwedge_{i=1}^{2} c_{i,k_i} \subseteq c_{1,1} \wedge c_{2,1}$ *for other choices of* k_i*'s; if indeed* $\bigwedge_{i=1}^{2} c_{i,k_i} \subseteq c_{1,1} \wedge c_{2,1}$ *for other choices of* k_i*'s, then only* 3 *subsumption testing is performed and we only keep the incomplete path* $c_{1,1} \rightarrow c_{2,1}$*. Likewise, for the segment* $\{C_3, C_4\}$*, the local subsumption testing may also detect that* $\bigwedge_{i=3}^{4} c_{i,k_i} \subseteq c_{3,1} \wedge c_{4,1}$ *by* 3 *subsumption testing, so that only the incomplete path* $c_{3,1} \rightarrow c_{4,1}$ *is kept. Thus in this scenario, local subsumtion testing with two segments (*$C_1 \wedge C_2$ *and* $C_3 \wedge C_4$*) may perform only* 6 *subsumption testing instead of 15 subsumption testing, hence has the potential to reduce the amount of runtime by reducing the number of possible subsumption testing.* □

In summary, we divide the whole conjunctive clauses $C_1, \ldots, C_m$ into a number of segments, and perform local subsumption testing to detect subsumption within each segment in advance, so that the overall number of subsumption testing may be improved. Note that different from the original subsumption testing in [63] that converts a polyhedral cone $P = \bigwedge_i c_{i,k_i}$ into the generated invariants I by taking the convex closure of P and I during the CNF-to-DNF expansion, the local subsumption testing in each segment cannot take the covex closure as it may cause inaccurate over-approximation to the original CNF. Thus, to keep the accuracy, in local subsumption testing we only have pairwise subsumption testing between different conjunctions expanded from the local segment (e.g., in the scenario of Figure 3 we have pairwise subsumption testing between $c_{1,1} \wedge c_{2,1}, c_{1,1} \wedge c_{2,2}, c_{1,2} \wedge c_{2,1}, c_{1,2} \wedge c_{2,2}$ within the segment $C_1 \wedge C_2$). In practice, we keep the size of each segment small to avoid the combinatorial explosion from the expansion of each segment.

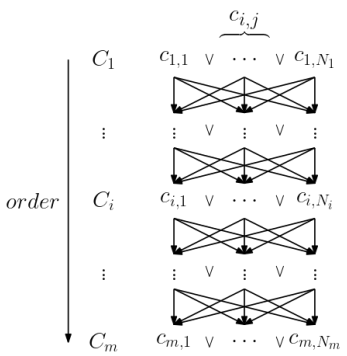

Fig. 2. CNF-to-DNF Expansion

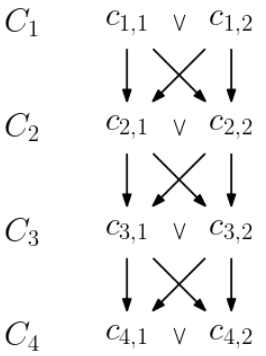

Fig. 3. Example of Expansion

Below we present the technical details of our improvement of segmented subsumption. The improvement comprises three steps (**Step B1** – **Step B3**) as follows.

Step B1 In the first step, we first reorder the CNF formula resulting from **Step A4** as in Section 3.1 and denote the reordered CNF as $\Phi = \bigwedge_{i=1}^{m} C_i$. We then divide the CNF formula Φ to be expanded into DNF into a number of segments. Our implementation divides the conjunctive clauses $C_1, \ldots, C_m$ into $\lceil m/k \rceil$ segments $C_1 \wedge \cdots \wedge C_k$, $\ldots$, $C_{i \cdot k+1} \wedge \cdots \wedge C_{i \cdot k+k}$, $\ldots$, $C_{(\lceil m/k \rceil - 1) \cdot k+1} \wedge \cdots \wedge C_m$, each with $k \geq 2$ conjunctive clauses except for the last segment. For example, when $m = 2 \cdot n$, we may choose $k = 2$ and divide $\bigwedge_{i=1}^{m} C_i$ into n segments $C_1 \wedge C_2$, $\ldots$, $C_{2n-1} \wedge C_{2n}$ Note that the conjunctive

clauses could also be segmented with different sizes, but for simplicity we choose a equal size (except for the last segment).

Step B2 In the second step, we perform local subsumption testing at every segment. Given a segment $C_p \wedge \cdots \wedge C_q$ with $C_i = c_{i,1} \vee \cdots \vee c_{i,N_i}$ for each $p \leq i \leq q$, the detailed local subsumption testing on this segment is as follows. We first expand the segment fully into its equivalent DNF

$$\bigvee_{1 \leq k_p \leq N_p, \cdots, 1 \leq k_q \leq N_q} Q_{k_p,\ldots,k_q} \text{ where } Q_{k_p,\ldots,k_q} := \bigwedge_{i=p}^{q} c_{i,k_i}. \tag{4}$$

Then for each pair of $Q_{k_p,\ldots,k_q}$ and $Q_{k'_p,\ldots,k'_q}$, we check whether one of them (e.g. $Q_{k_p,\ldots,k_q}$) is subsumed by the other (e.g. $Q_{k'_p,\ldots,k'_q}$); if the subsumption test is successful (e.g., $Q_{k_p,\ldots,k_q} \subseteq Q_{k'_p,\ldots,k'_q}$), then we remove the subsumed one (e.g., $Q_{k_p,\ldots,k_q}$) from the DNF. After this step, we get a DNF obtained from (4) by removing the subsumed $Q_{k_p,\ldots,k_q}$'s.

Step B3 In the last step, we group the DNFs obtained from **Step B2** for all segments conjunctively together to form a new CNF Φ'. (Note that this new CNF Φ' is logically equivalent to the original CNF Φ in **Step B1**.) We then run our invariant-generation procedure from the previous section (Section 4.2) (that incorporates various improvements related to the idea of generating the invariants one program location at a time) over the input Φ' to derive the invariants. Since Φ is logically equivalent to Φ', the obtained invariants are guaranteed to be correct.

The integration of the subsumption testing with our location-by-location idea is to first perform the reordered expansion in Section 4.1, then segmented subsumption testing, and finally apply the CNF-to-DNF expansion with projection-based generator computation from Section 4.2 to the CNF formula after segmented subsumption. The pseudo-code for our segmented subsumption testing is given in Algorithm 2 of Appendix A.

6 EXPERIMENTAL RESULTS

Implementation. We implemented our algorithms as an extension of StInG [71] in C++, and used PPL 1.2 [8] for polyhedra manipulation (e.g., projection, generation computation, etc.), and our algorithms work on the original input format of StInG [71]. To have a prototype implementation that synthesizes linear invariants directly over programs, we wrote most of the benchmark examples in C programming language by having statements that specify the change of values on each transitions, and implemented the transformation from these examples into the format in StInG [71] via Sparse 0.6.4 [70] (a C language semantic parser). An example input for one of our benchmarks (Scheduler) written in C is given in Appendix C. All experimental results were obtained on an Intel Core i7-7700 (3.6 GHz) machine with 15.5 GiB of memory, running Ubuntu 20.04.2 LTS.

Benchmarks. We consider benchmarks from a variety of application domains [5, 49, 63, 71, 72]:

Scheduler. The invariant analysis for cooperative multitasking (task scheduling) activated by interrupts and pre-emptive programming (in e.g. Arduino [5]) can be used to ensure the liveness of scheduling. We consider related benchmarks taken from [40, 63]. The benchmarks here are divided into two categories, namely "Scheduler'" and "Scheduler". The category "Scheduler'" contains original benchmarks from [63] (with original input format to StInG) that follow a non-standard setting to assign an initial condition to every location in the LinTS in order to model the scenario that the initial location is nondeterministic. The category "Scheduler" considers these benchmarks from StInG under the standard setting of a fixed initial location, corrects several places in the benchmarks that deviate from the original description in [40], and writes them in C.

Fischer. The Fischer mutual exclusion protocol [49] is a timing-based algorithm that implements mutual exclusion for a distributed system with skewed clocks, and the invariant analysis can be used

Table 3. Experimental Results for the Reordered CNF-to-DNF Expansion (†)

Benchmarks					StInG	Our Approach	
Name		Loc	Dim	Line		Reordered-Expansion†: Gen-Proj	
					Time (Banged)	Time (Banged)	Speedup
Scheduler'	2p	2	16	-	0.01 (26)	0.01 (34)	1.00X
	3p	3	33	-	0.17 (380)	0.10 (245)	1.70X
	4p	4	56	-	60.81 (26,629)	1.38 (1,508)	44.06X
	5p	5	85	-	7,436.34 (2,548,704)	26.52 (25,996)	280.40X
	6p	6	120	-	time out	892.75 (66,905)	>40.32X
	7p	7	161	-	time out	time out	-
Scheduler	3p	3	33	336	0.17 (279)	0.10 (261)	1.70X
	4p	4	56	609	4.16 (2,895)	0.98 (1,474)	4.24X
	5p	5	85	1017	135.80 (39,150)	13.97 (8,570)	9.72X
	6p	6	120	1587	7,541.53 (906,454)	277.96 (63,070)	27.13X
	7p	7	161	2352	time out	time out	-
Fischer	6p	7	63	710	9.18 (8,423)	3.07 (4,217)	2.99X
	7p	8	80	987	59.16 (32,668)	13.60 (12,566)	4.35X
	8p	9	99	1327	373.62 (127,918)	70.42 (43,191)	5.30X
	9p	10	120	1736	2,345.96 (503,369)	398.26 (163,623)	5.89X
	10p	11	143	2218	14,664.68 (1,985,857)	2,361.87 (649,409)	6.20X
	11p	12	168	2780	time out	14,307.40 (2,620,864)	>2.51X
	12p	13	195	3453	time out	time out	-
Cars	2p	3	27	216	0.01 (28)	0.01 (63)	1.00X
	3p	5	60	616	560.70 (788,508)	1.21 (2,299)	463.38X
	4p	7	105	1283	time out	86.83 (37,567)	>414.60X
	5p	9	162	2339	out of memory	out of memory	-

to adjust some parameters to ensure critical reachability properties. We take related benchmarks from the PAT toolkit [72] and rewrite them in C.

Cars. A scenario of car systems is illustrated as a dynamic decision problem [63] in which invariant analysis can be used to ensure the safety of autopilot. A car system has n cars on a straight road and their acceleration and velocity are determining by their controllers. The lead car starts at an arbitary acceleration with initial velocity 0. The controllers of other cars detect whether the distance between the front car and itself is too close or too far, and adjust their acceleration accordingly. We take the benchmarks from [63], modify them more naturally as LinTS with multiple locations (that distinguish different status on the distance between adjacent cars), and write them in C. We name the category of these benchmarks as "Cars".

For each benchmark above, we consider a variety on the number r of processes denoted by "r-p" (e.g. "2p" stands for two processes) in the benchmark. For all the benchmarks, we compare the running time between our algorithms and the original algorithm in StInG.

Experimental Results. Our experimental results are summarized in Table 3 – Table 6. Due to the limit of space, we only present the most representative experimental results and relegate others in Appendix B. Below we first describe the tables, then discuss the experimental results in detail.

Table Description. The detailed description of the tables is as follows. In all the tables, for the time consumption and the various size quantities of each benchmark, we have that "Loc" means the number of locations, "Dim" means the number of total dimensions (i.e., the number of all unknown coefficients at all locations), "Line" means the number of lines of the code representation in C, "Time" is the total amount of runtime (by summing up the time consumption at all locations) measured in seconds, "Banged" means the number of successful subsumptions (in the CNF-to-DNF expansion), "Our Approach" means the experimental results by our approach, "StInG" means the experimental results by StInG (which we adapt to PPL 1.2 [8]), the symbol "-" means not applicable due to either the absence of code representation or time-out or out-of-memory, and "Speedup" in each table shows the ratio of the amount of runtime consumed by StInG against our corresponding approach specified in the table. We set a time bound of 10 hours for time-out in all the tables.

Table 4. Experimental Results for Projection-Based Generator Computation

Benchmarks					Our Approach							
Name		Loc	Dim	Line	Reordered-Expansion†: Gen-Proj		Reordered-Expansion†: FME		Reordered-Expansion†: Block		Reordered-Expansion†: Block+	
					Time (Gen-Time)	Speedup	Time (Gen-Time)	Speedup	Time (Gen-Time)	Speedup	Time (Gen-Time)	Speedup
Scheduler'	2p	2	16	-	0.01 (0.01)	1.00X	0.01 (0.01)	1.00X	0.03 (0.02)	0.33X	0.02 (0.02)	0.50X
	3p	3	33	-	0.10 (0.06)	1.70X	0.14 (0.08)	1.21X	0.31 (0.27)	0.55X	0.14 (0.08)	1.21X
	4p	4	56	-	1.38 (0.24)	44.07X	1.60 (0.52)	38.01X	2.41 (1.42)	25.23X	1.50 (0.46)	40.54X
	5p	5	85	-	26.52 (0.38)	280.40X	27.11 (1.32)	274.30X	30.49 (4.03)	243.89X	27.27 (1.07)	272.69X
	6p	6	120	-	892.75 (38.59)	>40.32X	906.27 (48.88)	>39.72X	887.27 (28.01)	>40.57X	868.52 (11.20)	>41.45X
	7p	7	161	-	time out (61,366.73)	-	time out (66,745.07)	-	35,017.19 (209.25)	>1.03X	34,960.08 (140.49)	>1.03X
	8p	8	208	-	time out	-	time out	-	time out	-	time out	-
Scheduler	3p	3	33	336	0.10 (0.01)	1.70X	0.13 (0.05)	1.31X	0.26 (0.19)	0.65X	0.14 (0.07)	1.21X
	4p	4	56	609	0.98 (0.17)	4.24X	1.18 (0.39)	3.53X	1.96 (1.14)	2.12X	1.30 (0.37)	3.20X
	5p	5	85	1017	13.97 (1.22)	9.72X	14.58 (1.94)	9.31X	17.55 (4.91)	7.74X	18.68 (1.71)	7.27X
	6p	6	120	1587	277.96 (35.26)	27.13X	285.30 (45.20)	26.43X	262.31 (21.79)	28.75X	240.50 (6.60)	31.36X
	7p	7	161	2352	time out (61,660.65)	-	time out (66,808.22)	-	4,759.24 (142.23)	>7.56X	4,698.29 (77.68)	>7.66X
	8p	8	208	3351	time out	-	time out	-	time out	-	time out	-
Fischer	6p	7	63	710	3.07 (0.09)	2.99X	3.21 (0.40)	2.86X	3.44 (0.59)	2.67X	3.15 (0.21)	2.91X
	7p	8	80	987	13.60 (0.20)	4.35X	14.18 (0.69)	4.17X	14.73 (1.25)	4.02X	14.91 (0.74)	3.97X
	8p	9	99	1327	70.42 (0.35)	5.31X	71.97 (1.47)	5.19X	73.52 (2.66)	5.08X	71.15 (1.26)	5.25X
	9p	10	120	1736	398.26 (0.70)	5.89X	408.79 (2.69)	5.74X	413.76 (4.92)	5.67X	414.08 (2.59)	5.67X
	10p	11	143	2218	2,361.87 (1.34)	6.21X	2,429.26 (5.03)	6.04X	2,401.19 (8.75)	6.11X	2,398.38 (4.54)	6.11X
	11p	12	168	2780	14,307.40 (2.25)	>2.52X	14,469.76 (8.58)	>2.49X	14,483.90 (15.02)	>2.49X	14,318.84 (7.96)	>2.51X
	12p	13	195	3453	time out	-	time out	-	time out	-	time out	-
Cars	2p	3	27	216	0.01 (0)	1.00X	0.02 (0)	0.50X	0.02 (0)	0.50X	0.01 (0)	1.00X
	3p	5	60	616	1.21 (0.02)	463.39X	1.25 (0.01)	448.56X	1.55 (0.09)	361.74X	1.21 (0.04)	463.39X
	4p	7	105	1283	86.83 (0.11)	>414.60X	86.43 (0.58)	>416.52X	87.20 (1.29)	>412.84X	84.49 (0.56)	>426.09X
	5p	9	162	2339	out of memory	-	out of memory	-	out of memory	-	out of memory	-

Table 5. Experimental Results for Segmented Subsumption

Benchmarks					Our Approach(Reordered-Expansion†: Block+)							
Name		Loc	Dim	Line	none		Two		Three		Four	
					Time (Banged)	Speedup	Time (Banged)	Speedup	Time (Banged)	Speedup	Time (Banged)	Speedup
Scheduler'	2p	2	16	-	0.02 (34)	0.50X	0.03 (26)	0.33X	0.02 (38)	0.50X	0.02 (73)	0.50X
	3p	3	33	-	0.14 (245)	1.21X	0.20 (207)	0.85X	0.24 (195)	0.71X	0.23 (256)	0.74X
	4p	4	56	-	1.50 (1,508)	40.54X	1.84 (1,348)	33.05X	2.02 (1,301)	30.10X	2.69 (1,310)	22.61X
	5p	5	85	-	27.27 (25,996)	272.69X	20.86 (15,172)	356.49X	18.69 (9,991)	397.88X	24.35 (9,488)	305.39X
	6p	6	120	-	868.52 (66,905)	>41.45X	720.78 (54,862)	>49.95X	648.79 (46,226)	>55.49X	2,470.41 (39,207)	>14.57X
	7p	7	161	-	34,960.08 (629,977)	>1.03X	19,721.28 (437,681)	>1.83X	26,420.75 (449,445)	>1.36X	time out	-
	8p	8	208	-	time out	-	time out	-	time out	-	time out	-
Scheduler	3p	3	33	336	0.14 (261)	1.21X	0.19 (234)	0.89X	0.20 (212)	0.85X	0.21 (261)	0.81X
	4p	4	56	609	1.30 (1,474)	3.20X	1.78 (1,318)	2.34X	2.04 (1,270)	2.04X	3.59 (1,315)	1.16X
	5p	5	85	1017	18.68 (8,570)	7.27X	15.21 (6,728)	8.93X	37.73 (6,307)	3.60X	72.60 (6,065)	1.87X
	6p	6	120	1587	240.50 (63,070)	31.36X	201.53 (51,342)	37.42X	209.50 (42,837)	36.00X	596.82 (36,737)	12.64X
	7p	7	161	2352	4,698.29 (492,751)	>7.66X	2,752.66 (299,471)	>13.08X	3,760.55 (314,398)	>9.57X	8,986.91 (263,933)	>4.01X
	8p	8	208	3351	time out	-	time out	-	time out	-	time out	-
Fischer	6p	7	63	710	3.15 (4,217)	2.91X	3.19 (2,808)	2.88X	3.40 (2,336)	2.70X	4.40 (2,280)	2.09X
	7p	8	80	987	14.91 (12,566)	3.97X	9.45 (6,188)	6.26X	8.78 (4,407)	6.74X	10.87 (4,270)	5.44X
	8p	9	99	1327	71.15 (43,191)	5.25X	28.29 (14,231)	13.21X	25.39 (10,383)	14.72X	28.19 (8,690)	13.25X
	9p	10	120	1736	414.08 (163,623)	5.67X	91.96 (34,033)	25.51X	72.05 (22,273)	32.56X	75.95 (18,629)	30.89X
	10p	11	143	2218	2,398.38 (649,409)	6.11X	335.47 (90,362)	43.71X	185.65 (42,349)	78.99X	190.35 (36,250)	77.04X
	11p	12	168	2780	14,318.84 (2,620,864)	>2.51X	1,280.06 (235,812)	>28.12X	765.02 (129,144)	>47.06X	583.50 (81,593)	>61.70X
	12p	13	195	3453	time out	-	5,521.77 (681,865)	>6.52X	2,689.64 (296,297)	>13.38X	1,976.83 (190,150)	>18.21X
	13p	14	224	4397	time out	-	23,812.01 (1,835,173)	>1.51X	8,430.01 (588,289)	>4.27X	8,239.66 (501,438)	>4.37X
	14p	15	255	5039	time out	-	time out	-	time out	-	time out	-
Cars	2p	3	27	216	0.01 (63)	1.00X	0.04 (61)	0.25X	0.04 (62)	0.25X	0.05 (65)	0.20X
	3p	5	60	616	1.21 (2,299)	463.39X	2.01 (2,279)	278.96X	1.63 (1,592)	343.99X	2.04 (1,612)	274.85X
	4p	7	105	1283	84.49 (37,567)	>426.09X	100.35 (37,525)	>358.74X	89.78 (25,423)	>400.98X	103.59 (25,771)	>347.52X
	5p	9	162	2339	out of memory	-	out of memory	-	out of memory	-	out of memory	-

Table 3 presents the experimental results on our reordered CNF-to-DNF expansion (Section 4.1). We evaluate the expansion order (†) that expands (i) intra-transitions from the target location to itself first, then (ii) inter-transitions that involve the target location and one non-target location second, next (iii) the inter-transitions between different non-target locations third, and finally (iv) the intra-transitions over each individual non-target locations last. The expansion order follows the heuristics that the intra-transitions over the target location are the most relevant, and the counterpart over non-target locations are the least relevant. For the projection needed to produce the generators at the target location, we choose the basic method "Gen-Proj" that corresponds to the paragraph *Generator Projection* in Section 4.2.

Table 4 presents the experimental results on projection-based generator computation (Section 4.2) for which we use the expansion order in Table 3. Furthermore, the tags "Gen-Proj", "FME", "Block", "Block+" correspond to the methods from the paragraphs *Generator Projection*, *Fourier-Motzkin*

Table 6. Experimental Results for Parallel Computation

Benchmarks					Our Approach (Reordered-Expansion†: Block+)					
Name		Loc	Dim	Line	none			Two		
					Max	Speedup(original)	Speedup	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.01	0.50X	1.00X	0.02	0.33X	0.50X
	3p	3	33	-	0.05	1.21X	3.40X	0.08	0.85X	2.13X
	4p	4	56	-	0.47	40.54X	129.38X	0.56	33.05X	108.59X
	5p	5	85	-	7.52	272.69X	988.87X	7.33	356.49X	1,014.51X
	6p	6	120	-	308.48	>41.45X	>116.70X	302.22	>49.95X	>119.12X
	7p	7	161	-	11,134.30	>1.03X	>3.23X	5,944.38	>1.83X	>6.06X
Scheduler	3p	3	33	336	0.04	1.21X	4.25X	0.07	0.89X	2.43X
	4p	4	56	609	0.39	3.20X	10.66X	0.49	2.34X	8.49X
	5p	5	85	1017	4.90	7.27X	27.71X	4.42	8.93X	30.72X
	6p	6	120	1587	72.07	31.36X	104.64X	70.04	37.42X	107.67X
	7p	7	161	2352	1,336.13	>7.66X	>26.94X	750.41	>13.08X	>47.97X
Fischer	6p	7	63	710	1.22	2.91X	7.52X	0.79	2.88X	11.62X
	7p	8	80	987	7.75	3.97X	7.63X	2.85	6.26X	20.76X
	8p	9	99	1327	45.41	5.25X	8.22X	11.14	13.21X	33.54X
	9p	10	120	1736	283.95	5.67X	8.26X	43.96	25.51X	53.37X
	10p	11	143	2218	1,700.51	6.11X	8.62X	187.85	43.71X	78.07X
	11p	12	168	2780	10,230.51	>2.51X	>3.51X	768.56	>28.12X	>46.84X
	12p	13	195	3453	time out	-	-	3,437.67	>6.52X	>10.47X
	13p	14	224	4397	time out	-	-	14,885.30	>1.51X	>2.42X
Cars	2p	3	27	216	0.01	1.00X	1.00X	0.02	0.25X	0.50X
	3p	5	60	616	0.83	463.39X	675.54X	1.04	278.96X	539.13X
	4p	7	105	1283	75.83	>426.09X	>474.74X	84.46	>358.74X	>426.24X

Elimination, *Block Elimination* and *Our improvment on Block Elimination* in Section 4.2, respectively. The tag "Gen-Time" records the amount of time (in seconds) consumed on generator computation.

Table 5 presents the experimental results on segmented subsumption testing (Section 5) with our reordered expansion in Table 3 and "Block+" in Table 4. In the table, we use the tag "none" to indicate the absence of segmented subsumption, and the tags "Two" (resp. "Three", "Four") to indicate that the size of each segment is two (resp. three, four) except for the last segment, respectively.

Table 6 presents experimental results on parallel computation bestowed from our location-by-location idea. For the parallel computation, we assume the scenario that every location is assigned a distinct processor to generate the invariants at the location, so that the overall runtime here is the maximal runtime among all locations. In the table, we evaluate our approach with reordered expansion from Table 3 and "Block+" from Table 4, under the setting of both without (indicated by "none") and with the segmented subsumption testing of segment size two (indicated by "Two"); the column "Max" records the maximal runtime among all locations, the "Speedup" column shows the ratio of the amount of runtime by StInG against the "Max" column, and the "Speedup(original)" column records the ratio of the amount of runtime by StInG against the summation of runtime at at all locations (i.e., without parallelism).

Discussion. In all the benchmarks, the generated invariants by our approach (in each table) are exactly the same as from StInG, thus we focus on the comparison in runtime.

Table 3 compares the runtime under our reordered CNF-to-DNF expansion with the most basic projection method of "Gen-Proj" and without segmented subsumption testing. From the table, one can observe that our reordered expansion improves the performance of StInG up to orders of magnitude. (Note that the speedup value with > means that StInG times out on the benchmark). This demonstrates that our reordered expansion can indeed improve the scalability. The improvement can also be observed by the fact that the expansion reordering induces a smaller number of successful subsumptions, which means that these subsumptions are detected earlier by the reordering.

Based on our reordered CNF-to-DNF expansion, Table 4 further compares the runtime from various projection-based generator-computation methods. From the table, one can observe that the simplest way "Gen-Proj" suffices to have comparable speedup against other methods when the dimension is moderate. However, for several high dimensional examples (such as Scheduler-7p and Scheduler-7p'), "Block" and "Block+" are the only methods that help complete the invariant generation in tractable time. Moreover, when comparing "Block" and "Block+", one observes that "Block+" consistently improves "Block" in the runtime for generator computation. This shows that our "Block+" improvement can further leverage the scalability for high dimensional examples in addition to reordered expansion.

Table 5 shows that in addition to our reordered CNF-to-DNF expansion (†) and "Block+", segmented subsumption testing is effective in high dimensional examples, such as Scheduler-7p, Scheduler'-7p and from Fischer-8p to Fischer-13p, especially for the Fischer benchmarks where a large amount of local subsumption exists, segmented subsumption testing makes it possible to scale the invariant generation to Fischer-12p and Fischer-13p. However, incorporating segmented subsumption is not always better since itself incurs extra runtime at local subsumption testing.

Finally, Table 6 checks the advantage of parallel processing endowed by our location-by-location idea implemented with our reordered CNF-to-DNF expansion (†) and "Block+". Under the setting of with and without segmented subsumption. From the experimental data, one observes that parallel processing further increases the scalability by a factor between 2 and 3.

Note that our approach cannot scale beyond Cars-4p since we encounter out-of-memory, although the runtime for Cars-2p, Cars-3p and Cars-4p is relatively low. Since our implementation is an extension of StInG, we believe that by improving the memory management in StInG, our approach could handle larger examples from this class of benchmarks.

Remark 2. *Here we compare our experiment results with abstract interpretation and recurrence analysis. On one hand, in [63], abstract interpretation with standard polyhedral abstract domain [7, 23] has been compared with the approach [63] that implements Farkas' Lemma with several reasonale heuristics; the comparison was that the approach [63] (with heuristics that incurs accuracy loss) often generates much more accurate invariants than abstract interpretation and can handle examples where abstract interpretation seemingly diverges. Note that recent advances [69] in polyhedral abstract domain explores possible speed up through the separation of independent variables (which is not the case in our benchmarks since all variables in our benchmarks are correlated) , but do not improve the accuracy [69]. On the other hand, the state of the art recurrence solver OCRS [55] can only generate invariants that relates the value of a program variable and the loop counter, thus cannot be applied to our benchmarks.* □

Remark 3. *We remark that popular software verification frameworks such as CPAchecker [24], SeaHorn [65] and Ultimate Automizer [74] only consider to check the validity of a given assertion at a program location, thus invariant generation in these frameworks requires an additional assertion as the post-condition and only aims at generating enough invariants to prove/disprove the assertion. In contrast, we follow the setting in [18, 23, 40, 63] that does not require post-condition and aims at generating as much invariants as possible. Thus, these frameworks do not apply to our case. Nevertheless, our improvements can still be applied to the case with assertions, as one can focus on the target program location where the assertion lies, and perform segmented subsumption to simplify the CNF formula.* □

7 RELATED WORKS

Below we compare our approach with most related approaches on numerical invariant generation. We present the comparison classifying the related approaches into different categories of methods.

Constraint Solving. Our approach falls in this category. Since we focus on linear invariant generation, our approach is incomparable with those for polynomial invariant generation [1, 14, 16, 20, 27, 42, 43, 45, 50, 60, 62, 76]. Below we compare our approach with the approaches for linear invariant generation in constraint solving [18, 28, 39, 63]. The approach [18] first establishes the framework of linear invariant generation through Farkas' Lemma and solves the invariant through the complete method of quantifier elimination. Quantifier elimination usually has high runtime complexity and is impractical even for programs in moderate size [78]. Thus, our approach has much better runtime performance by developing various techniques to improve scalability.

The approach [63] considers several heuristics to solve the non-linear constraints from the application of Farkas' Lemma, hence is more scalable than [18]. A main disadvantage of [63] is that it solves the invariants at all program locations in a single invariant-generation process, thus causing a potentially large amount of combinatorial explosion. Our approach is based on [63] and further incoporates (i) the novel idea of handling the program locations one by one with reinforcing technical improvements and (ii) the segmented subsumption testing to further discover subsumption in the CNF-to-DNF expansion, thus can substantially mitigate the combinatorial explosion from [63] and achieve much better scalability.

The approach [28] uses eigenvectors to handle several restricted classes of linear invariants. Our approach tackles the general class of affine programs and invariants through completely different techniques, thus focuses on a completely different aspect. The tool INVGEN [39] focuses on how one integrates abstract interpretation and constraint solving, while our approach aims at improving scalability of constraint solving. Thus the focuses of our approach and INVGEN are orthogonal.

Abstract Interpretation. The most classical method to find inductive invariants is *abstract interpretation* [2, 9, 12, 22, 53, 59, 61]. An abstraction-interpretation based approach follows the paradigm of first having an abstract domain for the desired invariants, and then perform forward propagation with widening to reach a fixed point. Compared with constraint solving, abstract interpretation does not provide guarantee on the accuracy of the generated invaraints, except for some rare cases [37]. This point is experimental observed in [63], where the method of constraint solving (even with some reasonable heuristics that cause accuracy loss) can produce much tighter invariants than an abstract interpretation through polyhedral abstract domain. We prove that our approach generates the same invariants as [63], hence inherits the advantage of high accuracy from constraint solving.

Recurrence Analysis. Recurrence analysis [11, 31, 44, 47, 48] first transforms the invariant-generation problem into a recurrence relation, and then solves the invariants through the analysis of the recurrence relation. Compared with constraint solving, recurrence-based approaches are only applicable to the situations where the underlying recurrence relation has a closed form solution, while constraint solving can be applied when a closed form solution does not exist.

Other Methods. Invariant could also be generated through logical inference [29, 34, 35, 38, 52, 66, 75], machine learning [36, 41, 77] and dynamic analysis [25, 54, 68]. Since these methods either utilize specilized inference rules, or depend on a potentially big dataset, or require a possible large amount of program executions, they could not ensure any accuracy guarantee on the generated invariants.

8 CONCLUSION AND FUTURE WORK

We proposed two improvements to the previous approach [63] that solves the invariant generation problem with Farkas' Lemma and several reasonable heuristics. The first is a novel idea that generates the linear invariants for each program location separately. The idea enables one to speed up the invariant-generation process by parallel processing, and we reinforce the idea with two technical improvments that further reduce the combinatorial explosion from [63]: (i) in the CNF-to-DNF expansion, we reordered the expansion so that the target location (over which the invariants

are to be generated) comes first to detect subsumptions earlier; (ii) in the generator computation, we proposed various methods that calculate the generators projected onto the target location, including a novel method that improves Block Elimination. The second is a segmented subsumption testing in addition to the subsumption testing in [63] that could detect subsumptions at local conjunctive clauses. Experimental results show that our improvements can improve the scalability of the approach [63], the only known practical algorithm that is based on Farkas' Lemma, by orders of magnitude. A future direction would be to investigate new approaches to further improve scalability. Another future direction would be to extend our improvements to polynomial invariant generation [14]. A further direction is to integrate the approach in [67] to handle disjunctive invariants.

REFERENCES

[1] Assalé Adjé, Pierre-Loïc Garoche, and Victor Magron. 2015. Property-based Polynomial Invariant Generation Using Sums-of-Squares Optimization. In *SAS (LNCS, Vol. 9291)*. Springer, 235–251.

[2] Assalé Adjé, Stéphane Gaubert, and Eric Goubault. 2012. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. *Log. Methods Comput. Sci.* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:1)2012

[3] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012. Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification. In *CAV (LNCS, Vol. 7358)*. Springer, 672–678. https://doi.org/10.1007/978-3-642-31424-7_48

[4] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS (LNCS, Vol. 6337)*. Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

[5] Arduino [n.d.]. Arduino: An open-source electronics platform based on easy-to-use hardware and software. https://github.com/arkhipenko/TaskScheduler.

[6] Ali Asadi, Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Mohammad Mahdavi. 2021. Polynomial reachability witnesses via Stellensätze. In *PLDI*. ACM, 772–787. https://doi.org/10.1145/3453483.3454076

[7] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. 2003. Precise Widening Operators for Convex Polyhedra. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 337–354. https://doi.org/10.1007/3-540-44898-5_19

[8] Roberto Bagnara, Elisa Ricci, Enea Zaffanella, and Patricia M. Hill. 2002. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *SAS (Lecture Notes in Computer Science, Vol. 2477)*. Springer, 213–229. https://doi.org/10.1007/3-540-45789-5_17

[9] Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. 2005. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In *SAS (LNCS, Vol. 3672)*. Springer, 19–34. https://doi.org/10.1007/11547662_4

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV (LNCS, Vol. 3576)*. Springer, 491–504. https://doi.org/10.1007/11513988_48

[11] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas W. Reps. 2020. Templates and recurrences: better together. In *PLDI*. ACM, 688–702. https://doi.org/10.1145/3385412.3386035

[12] Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *SAS (LNCS, Vol. 8723)*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer, 85–100. https://doi.org/10.1007/978-3-319-10936-7_6

[13] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial Worst-Case Analysis of Recursive Programs. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 20:1–20:52. https://doi.org/10.1145/3339984

[14] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *PLDI*. ACM, 672–687. https://doi.org/10.1145/3385412.3385969

[15] Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017. Stochastic invariants for probabilistic termination. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 145–160. https://doi.org/10.1145/3009837.3009873

[16] Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *CAV (LNCS, Vol. 9206)*. Springer, 658–674. https://doi.org/10.1007/978-3-319-21690-4_44

[17] Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. 2007. Discovering Non-linear Ranking Functions by Solving Semi-algebraic Systems. In *ICTAC (LNCS, Vol. 4711)*. Springer, 34–49. https://doi.org/10.1007/978-3-540-75292-9_3

[18] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV (LNCS, Vol. 2725)*. Springer, 420–432. https://doi.org/10.1007/978-3-540-45069-6_39

[19] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *TACAS (LNCS, Vol. 2031)*. Springer, 67–81. https://doi.org/10.1007/3-540-45319-9_6

[20] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI (LNCS, Vol. 3385)*. Springer, 1–24. https://doi.org/10.1007/978-3-540-30579-8_1

[21] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252. https://doi.org/10.1145/512950.512973

[22] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *ESOP (LNCS, Vol. 3444)*. Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[23] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*. ACM Press, 84–96. https://doi.org/10.1145/512760.512770
[24] CPAchecker [n.d.]. CPAchecker: The Configurable Software-Verification Platform. https://cpachecker.sosy-lab.org.
[25] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: dynamic symbolic execution for invariant inference. In *ICSE*. ACM, 281–290. https://doi.org/10.1145/1368088.1368127
[26] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. 2016. Danger Invariants. In *FM (LNCS, Vol. 9995)*. 182–198. https://doi.org/10.1007/978-3-319-48989-6_12
[27] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial Invariants by Linear Algebra. In *ATVA (LNCS, Vol. 9938)*. 479–494. https://doi.org/10.1007/978-3-319-46520-3_30
[28] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2017. Synthesizing Invariants by Solving Solvable Loops. In *ATVA (LNCS, Vol. 10482)*. Springer, 327–343. https://doi.org/10.1007/978-3-319-68167-2_22
[29] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. ACM, 443–456. https://doi.org/10.1145/2509136.2509511
[30] J. Farkas. 1894. A Fourier-féle mechanikai elv alkalmazásai (Hungarian). *Mathematikaiés Természettudományi Értesitö* 12 (1894), 457–472.
[31] Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*. IEEE, 57–64.
[32] Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *ATVA (LNCS, Vol. 10482)*. Springer, 400–416. https://doi.org/10.1007/978-3-319-68167-2_26
[33] Komei Fukuda and Alain Prodon. 1995. Double Description Method Revisited. In *Combinatorics and Computer Science, 8th Franco-Japanese and 4th Franco-Chinese Conference, Brest, France, July 3-5, 1995, Selected Papers (Lecture Notes in Computer Science, Vol. 1120)*, Michel Deza, Reinhardt Euler, and Yannis Manoussakis (Eds.). Springer, 91–111. https://doi.org/10.1007/3-540-61576-8_77
[34] Ting Gan, Bican Xia, Bai Xue, Naijun Zhan, and Liyun Dai. 2020. Nonlinear Craig Interpolant Generation. In *CAV (LNCS, Vol. 12224)*. Springer, 415–438. https://doi.org/10.1007/978-3-030-53288-8_20
[35] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV (LNCS, Vol. 8559)*. Springer, 69–87. https://doi.org/10.1007/978-3-319-08867-9_5
[36] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL*. ACM, 499–512. https://doi.org/10.1145/2837614.2837664
[37] Roberto Giacobazzi and Francesco Ranzato. 1997. Completeness in Abstract Interpretation: A Domain Perspective. In *AMAST (LNCS, Vol. 1349)*. Springer, 231–245. https://doi.org/10.1007/BFb0000474
[38] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2009. Constraint-Based Invariant Inference over Predicate Abstraction. In *VMCAI (LNCS, Vol. 5403)*. Springer, 120–135. https://doi.org/10.1007/978-3-540-93900-9_13
[39] Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *CAV (LNCS, Vol. 5643)*. Springer, 634–640. https://doi.org/10.1007/978-3-642-02658-4_48
[40] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. 1997. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods Syst. Des.* 11, 2 (1997), 157–185. https://doi.org/10.1023/A:1008678014487
[41] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2020. Learning fast and precise numerical analysis. In *PLDI*. ACM, 1112–1127. https://doi.org/10.1145/3385412.3386016
[42] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2018. Polynomial Invariants for Affine Programs. In *LICS*. ACM, 530–539. https://doi.org/10.1145/3209108.3209142
[43] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017. Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In *ISSAC*. ACM, 221–228. https://doi.org/10.1145/3087604.3087623
[44] Andreas Humenberger and Laura Kovács. 2021. Algebra-Based Synthesis of Loops and Their Invariants (Invited Paper). In *VMCAI (LNCS, Vol. 12597)*. Springer, 17–28. https://doi.org/10.1007/978-3-030-67067-2_2
[45] Deepak Kapur. 2005. Automatically Generating Loop Invariants Using Quantifier Elimination. In *Deduction and Applications (Dagstuhl Seminar Proceedings, Vol. 05431)*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/511
[46] Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *SAS (LNCS, Vol. 6337)*. Springer, 390–406. https://doi.org/10.1007/978-3-642-15769-1_24
[47] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *PLDI*. ACM, 248–262. https://doi.org/10.1145/3062341.3062373
[48] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. *Proc. ACM Program. Lang.* 2, POPL (2018), 54:1–54:33. https://doi.org/10.1145/3158142
[49] Leslie Lamport. 1987. A Fast Mutual Exclusion Algorithm. *ACM Trans. Comput. Syst.* 5, 1 (1987), 1–11. https://doi.org/10.1145/7351.7352

[50] Wang Lin, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. 2014. Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. *Frontiers Comput. Sci.* 8, 2 (2014), 192–202.

[51] Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems - safety*. Springer.

[52] Kenneth L. McMillan. 2008. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *TACAS (LNCS, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 413–427. https://doi.org/10.1007/978-3-540-78800-3_31

[53] Markus Müller-Olm and Helmut Seidl. 2004. Computing polynomial program invariants. *Inf. Process. Lett.* 91, 5 (2004), 233–244. https://doi.org/10.1016/j.ipl.2004.05.004

[54] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*. IEEE Computer Society, 683–693. https://doi.org/10.1109/ICSE.2012.6227149

[55] OCRS [n.d.]. OCRS: Operational calculus recurrence solver. https://github.com/cyphertjohn/OCRS.

[56] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM, 614–630. https://doi.org/10.1145/2908080.2908118

[57] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI (LNCS, Vol. 2937)*. Springer, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

[58] PPL 2021. Parma Polyhedra Library, PPL 1.2. https://www.bugseng.com/parma-polyhedra-library.

[59] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *SAS (LNCS, Vol. 3148)*. Springer, 280–295. https://doi.org/10.1007/978-3-540-27864-1_21

[60] Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *ISSAC*. ACM, 266–273. https://doi.org/10.1145/1005285.1005324

[61] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.* 64, 1 (2007), 54–75. https://doi.org/10.1016/j.scico.2006.03.003

[62] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using Gröbner bases. In *POPL*. ACM, 318–329. https://doi.org/10.1145/964001.964028

[63] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2004. Constraint-Based Linear-Relations Analysis. In *SAS (LNCS, Vol. 3148)*. Springer, 53–68. https://doi.org/10.1007/978-3-540-27864-1_7

[64] Alexander Schrijver. 1999. *Theory of linear and integer programming*. Wiley.

[65] SeaHorn [n.d.]. SeaHorn: A fully automated analysis framework for LLVM-based languages. http://seahorn.github.io.

[66] Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods Syst. Des.* 48, 3 (2016), 235–256. https://doi.org/10.1007/s10703-016-0248-5

[67] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying Loop Invariant Generation Using Splitter Predicates. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 703–719. https://doi.org/10.1007/978-3-642-22110-1_57

[68] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP (LNCS, Vol. 7792)*. Springer, 574–592. https://doi.org/10.1007/978-3-642-37036-6_31

[69] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59.

[70] Sparse [n.d.]. Sparse: C language semantic parser. https://lwn.net/Articles/689907/.

[71] StInG [n.d.]. StInG: Stanford Invariant Generator. http://theory.stanford.edu/~srirams/Software/sting.html.

[72] Jun Sun, Yang Liu, Jin Song Dong, and Xian Zhang. 2009. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5885)*, Karin K. Breitman and Ana Cavalcanti (Eds.). Springer, 581–600. https://doi.org/10.1007/978-3-642-10373-5_30

[73] Delaram Talaashrafi. 2018. *Complexity Results for Fourier-Motzkin Elimination (Thesis format: Monograph)*. Ph.D. Dissertation. The University of Western Ontario London.

[74] UltimateAutomizer [n.d.]. UltimateAutomizer: A Software Model Checker. https://monteverdi.informatik.uni-freiburg.de/tomcat/Website/?ui=tool&tool=automizer.

[75] Rongchen Xu, Fei He, and Bow-Yaw Wang. 2020. Interval counterexamples for loop invariant learning. In *ESEC/FSE*. ACM, 111–122. https://doi.org/10.1145/3368089.3409752

[76] Lu Yang, Chaochen Zhou, Naijun Zhan, and Bican Xia. 2010. Recent advances in program verification through computer algebra. *Frontiers Comput. Sci. China* 4, 1 (2010), 1–16. https://doi.org/10.1007/s11704-009-0074-7

[77] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *PLDI*. ACM, 106–120. https://doi.org/10.1145/3385412.3385986

[78] Hengjun Zhao, Naijun Zhan, Deepak Kapur, and Kim G. Larsen. 2012. A "Hybrid" Approach for Synthesizing Optimal Controllers of Hybrid Systems: A Case Study of the Oil Pump Industrial Example. In *FM 2012: Formal Methods - 18th*

International Symposium, Paris, France, August 27-31, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7436), Dimitra Giannakopoulou and Dominique Méry (Eds.). Springer, 471–485.

A PSEUDO-CODE FOR SEGMENTED SUBSUMPTION TESTING

Algorithm 2 Segmented Subsumption Testing

Input: $\bigwedge_{i=1}^{m} C_i$: the CNF formula obtained from **Step A4** (Section 3.1) in the approach [Colón et al. 2003] where each $C_i = \bigvee_{j=1}^{N_i} c_{i,j}$ is the i-th conjunctive clause and is a disjunction of polyhedra over the unknown coefficients in the template;
k : the size of segments (the size could be different but for simplicity we choose a equal size except for the last segment)

Output: Φ' : the segmented CNF formula

1: $\{\phi_0, \cdots, \phi_{(\lceil m/k \rceil - 1)}\} \leftarrow$ divide $\bigwedge_{i=1}^{m} C_i$ into $\lceil m/k \rceil$ segments;
   //$\phi_i = C_p \wedge \cdots \wedge C_q$ where $p = i \cdot k + 1$ and $q = i \cdot k + k$ (except $q = m$ if $i = \lceil m/k \rceil - 1$)
2: $\Phi_i \leftarrow \emptyset$ (for $0 \leq i \leq \lceil m/k \rceil - 1$);
3: **for** $i = 0$ to $\lceil m/k \rceil - 1$ **do**
4: $\quad \bigvee Q_{k_p,\ldots,k_q} \leftarrow$ expand ϕ_i fully into its equivalent DNF;
   $\quad$ //$Q_{k_p,\ldots,k_q} = \bigwedge_{i=p}^{q} c_{i,k_i}$ where $1 \leq k_p \leq n_p, \cdots, 1 \leq k_q \leq n_q$
5: $\quad$ **for each** $Q_{k_p,\ldots,k_q}$ **and** $Q_{k'_p,\ldots,k'_q}$ such that $Q_{k_p,\ldots,k_q} \neq Q_{k'_p,\ldots,k'_q}$ **do**
   $\quad\quad$ //$1 \leq k_i, k'_i \leq N_i$ where $p \leq i \leq q$
6: $\quad\quad$ **if** $Q_{k_p,\ldots,k_q} \subseteq Q_{k'_p,\ldots,k'_q}$ **then**
7: $\quad\quad\quad$ remove subsumed $Q_{k_p,\ldots,k_q}$ from $\bigvee Q_{k_p,\ldots,k_q}$;
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: $\quad \Phi_i \leftarrow \bigvee Q_{k_p,\ldots,k_q}$;
11: **end for**
12: $\Phi' \leftarrow \Phi_0 \bigwedge \cdots \bigwedge \Phi_{(\lceil m/k \rceil - 1)}$;
13: **return** Φ';

B DETAILED EXPERIMENTAL RESULTS

Table 7. Experimental Results for Parallel Computation (Reordered-Expansion: Gen-Proj)

Benchmarks					Time (Reordered-Expansion: Gen-Proj)														
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.01	0	-	-	-	-	-	-	-	-	-	-	0.01	1.00X	1.00X
	3p	3	33	-	0.04	0.03	0.03	-	-	-	-	-	-	-	-	-	0.04	1.70X	4.25X
	4p	4	56	-	0.40	0.45	0.32	0.21	-	-	-	-	-	-	-	-	0.45	44.06X	135.13X
	5p	5	85	-	6.93	5.81	7.36	4.54	1.88	-	-	-	-	-	-	-	7.36	280.40X	1010.37X
	6p	6	120	-	42.30	178.32	310.36	216.76	114.69	30.32	-	-	-	-	-	-	310.36	>40.32X	>115.99X
	7p	7	161	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Scheduler	3p	3	33	336	0.03	0.04	0.03	-	-	-	-	-	-	-	-	-	0.04	1.70X	4.25X
	4p	4	56	609	0.23	0.27	0.26	0.22	-	-	-	-	-	-	-	-	0.27	4.24X	15.40X
	5p	5	85	1017	2.40	3.29	4.16	2.83	1.29	-	-	-	-	-	-	-	4.16	9.72X	32.64X
	6p	6	120	1587	27.75	47.90	80.32	55.99	44.81	21.19	-	-	-	-	-	-	80.32	27.13X	93.89X
	7p	7	161	2352	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Fischer	6p	7	63	710	0.40	0.19	0.20	0.26	0.47	1.21	0.34	-	-	-	-	-	1.21	2.99X	7.58X
	7p	8	80	987	0.90	0.44	0.45	0.53	0.85	2.17	7.40	0.86	-	-	-	-	7.40	4.35X	7.99X
	8p	9	99	1327	1.78	0.94	0.98	1.12	1.67	3.89	12.34	45.49	2.21	-	-	-	45.49	5.30X	8.21X
	9p	10	120	1736	3.31	1.89	1.95	2.22	3.33	7.15	21.29	74.17	277.24	5.71	-	-	277.24	5.89X	8.46X
	10p	11	143	2218	5.97	3.66	3.82	4.38	6.45	13.68	38.88	128.98	456.91	1,684.68	14.46	-	1,684.68	6.20X	8.70X
	11p	12	168	2780	10.42	6.96	7.25	8.55	12.97	27.77	76.58	240.98	813.28	2,841.46	10,224.63	36.55	10,224.63	>2.51X	>3.52X
Cars	2p	3	27	216	0.01	<0.01	<0.01	-	-	-	-	-	-	-	-	-	0.01	1.00X	1.00X
	3p	5	60	616	0.84	0.08	0.09	0.10	0.10	-	-	-	-	-	-	-	0.84	463.38X	667.50X
	4p	7	105	1283	76.99	1.55	1.48	1.69	1.67	1.72	1.73	-	-	-	-	-	76.99	>414.60X	>467.59X

Table 8. Experimental Results for Parallel Computation (Reordered-Expansion: FME)

Benchmarks					Time (Reordered-Expansion: FME)														
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.01	<0.01	-	-	-	-	-	-	-	-	-	-	0.01	1.00X	1.00X
	3p	3	33	-	0.05	0.04	0.05	-	-	-	-	-	-	-	-	-	0.05	1.21X	3.40X
	4p	4	56	-	0.48	0.50	0.36	0.26	-	-	-	-	-	-	-	-	0.50	38.00X	121.62X
	5p	5	85	-	7.04	5.93	7.46	4.66	2.02	-	-	-	-	-	-	-	7.46	274.30X	996.82X
	6p	6	120	-	42.26	181.22	313.45	220.01	116.21	33.12	-	-	-	-	-	-	313.45	>39.72X	>114.85X
	7p	7	161	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Scheduler	3p	3	33	336	0.04	0.05	0.04	-	-	-	-	-	-	-	-	-	0.05	1.30X	3.40X
	4p	4	56	609	0.29	0.32	0.30	0.27	-	-	-	-	-	-	-	-	0.32	3.52X	13.00X
	5p	5	85	1017	2.51	3.44	4.30	2.91	1.42	-	-	-	-	-	-	-	4.30	9.31X	31.58X
	6p	6	120	1587	28.43	51.57	80.06	58.66	43.09	23.49	-	-	-	-	-	-	80.06	26.43X	94.19X
	7p	7	161	2352	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Fischer	6p	7	63	710	0.41	0.23	0.23	0.28	0.46	1.25	0.35	-	-	-	-	-	1.25	2.85X	7.34X
	7p	8	80	987	0.91	0.51	0.52	0.60	0.93	2.23	7.54	0.94	-	-	-	-	7.54	4.17X	7.84X
	8p	9	99	1327	1.83	1.06	1.09	1.23	1.80	4.03	12.48	46.06	2.39	-	-	-	46.06	5.19X	8.11X
	9p	10	120	1736	3.55	2.11	2.18	2.46	3.51	7.36	21.81	75.38	284.34	6.09	-	-	284.34	5.73X	8.25X
	10p	11	143	2218	6.43	4.03	4.22	4.81	7.05	14.18	40.05	131.75	465.63	1,736.17	14.94	-	1,736.17	6.03X	8.44X
	11p	12	168	2780	10.96	7.47	7.83	9.11	13.64	28.67	78.04	244.84	822.43	2,869.50	10,339.85	37.42	10,339.85	>2.48X	>3.48X
Cars	2p	3	27	216	0.01	0.01	<0.01	-	-	-	-	-	-	-	-	-	0.01	0.50X	1.00X
	3p	5	60	616	0.85	0.09	0.09	0.11	0.11	-	-	-	-	-	-	-	0.85	448.56X	659.64X
	4p	7	105	1283	77.60	1.36	1.32	1.52	1.52	1.56	1.55	-	-	-	-	-	77.60	>416.52X	>463.91X

Table 9. Experimental Results for Parallel Computation (Reordered-Expansion: Block)

Benchmarks					Time (Reordered-Expansion: Block)														
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.01	0.02	-	-	-	-	-	-	-	-	-	-	0.02	0.33X	0.50X
	3p	3	33	-	0.11	0.10	0.10	-	-	-	-	-	-	-	-	-	0.11	0.54X	1.54X
	4p	4	56	-	0.65	0.69	0.59	0.48	-	-	-	-	-	-	-	-	0.69	25.23X	88.13X
	5p	5	85	-	7.68	6.62	8.21	5.34	2.64	-	-	-	-	-	-	-	8.21	243.89X	905.76X
	6p	6	120	-	44.24	177.94	313.50	220.60	112.74	18.25	-	-	-	-	-	-	313.50	>40.57X	>114.83X
	7p	7	161	-	940.00	5,756.17	11,109.94	9,211.10	5,232.45	2,660.25	107.28	-	-	-	-	-	11,109.94	>1.03X	>3.24X
Scheduler	3p	3	33	336	0.08	0.09	0.09	-	-	-	-	-	-	-	-	-	0.09	0.65X	1.88X
	4p	4	56	609	0.46	0.52	0.49	0.49	-	-	-	-	-	-	-	-	0.52	2.12X	8.00X
	5p	5	85	1017	3.11	4.07	4.87	3.50	2.00	-	-	-	-	-	-	-	4.87	7.73X	27.88X
	6p	6	120	1587	30.10	47.30	75.93	59.12	41.39	8.47	-	-	-	-	-	-	75.93	28.75X	99.32X
	7p	7	161	2352	452.42	778.46	1,340.36	893.81	746.18	506.50	41.51	-	-	-	-	-	1,340.36	>7.56X	>26.86X
Fischer	6p	7	63	710	0.43	0.25	0.27	0.30	0.50	1.29	0.40	-	-	-	-	-	1.29	2.66X	7.11X
	7p	8	80	987	0.96	0.58	0.59	0.67	1.00	2.30	7.61	1.02	-	-	-	-	7.61	4.01X	7.77X
	8p	9	99	1327	1.96	1.18	1.21	1.37	1.93	4.15	12.87	46.34	2.51	-	-	-	46.34	5.08X	8.06X
	9p	10	120	1736	3.74	2.38	2.38	2.67	3.76	7.68	22.17	76.12	286.63	6.23	-	-	286.63	5.66X	8.18X
	10p	11	143	2218	6.68	4.39	4.51	5.09	7.23	14.63	40.31	132.17	465.49	1,705.49	15.20	-	1,705.49	6.10X	8.59X
	11p	12	168	2780	11.44	8.01	8.34	9.66	14.19	29.08	78.58	246.01	823.62	2,871.39	10,345.63	37.95	10,345.63	>2.48X	>3.47X
Cars	2p	3	27	216	0.01	0.01	0	-	-	-	-	-	-	-	-	-	0.01	0.50X	1.00X
	3p	5	60	616	1.01	0.12	0.12	0.15	0.15	-	-	-	-	-	-	-	1.01	361.74X	555.14X
	4p	7	105	1283	77.75	1.46	1.42	1.62	1.62	1.67	1.66	-	-	-	-	-	77.75	>412.84X	>463.02X

Table 10. Experimental Results for Parallel Computation (Reordered-Expansion: Block+)

Benchmarks					Time (Reordered-Expansion: Block+)														
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.01	0.01	-	-	-	-	-	-	-	-	-	-	0.01	0.50X	1.00X
	3p	3	33	-	0.05	0.04	0.05	-	-	-	-	-	-	-	-	-	0.05	1.21X	3.40X
	4p	4	56	-	0.41	0.47	0.36	0.26	-	-	-	-	-	-	-	-	0.47	40.54X	129.38X
	5p	5	85	-	7.04	6.01	7.52	4.7	2	-	-	-	-	-	-	-	7.52	272.69X	988.87X
	6p	6	120	-	41.77	175.8	308.48	216.41	110.84	15.22	-	-	-	-	-	-	308.48	>41.45X	>116.70X
	7p	7	161	-	928.54	5,755.83	11,134.30	9,202.96	5,199.67	2,646.72	92.06	-	-	-	-	-	11,134.30	>1.03X	>3.23X
Scheduler	3p	3	33	336	0.04	0.04	0.04	-	-	-	-	-	-	-	-	-	0.04	1.21X	4.25X
	4p	4	56	609	0.32	0.39	0.31	0.26	-	-	-	-	-	-	-	-	0.39	3.20X	10.66X
	5p	5	85	1017	2.63	3.54	4.90	3.37	1.38	-	-	-	-	-	-	-	4.90	7.27X	27.71X
	6p	6	120	1587	28.74	44.04	72.07	51.77	38.62	5.26	-	-	-	-	-	-	72.07	31.36X	104.64X
	7p	7	161	2352	440.33	776.02	1,336.13	887.65	735.75	495.42	26.99	-	-	-	-	-	1,336.13	>7.66X	>26.94X
Fischer	6p	7	63	710	0.40	0.22	0.23	0.26	0.44	1.22	0.38	-	-	-	-	-	1.22	2.91X	7.52X
	7p	8	80	987	0.92	0.53	0.54	0.62	1.00	2.53	7.75	1.02	-	-	-	-	7.75	3.97X	7.63X
	8p	9	99	1327	1.86	1.04	1.07	1.21	1.79	3.97	12.45	45.41	2.35	-	-	-	45.41	5.25X	8.22X
	9p	10	120	1736	3.48	2.10	2.12	2.40	3.48	7.71	25.06	77.82	283.95	5.96	-	-	283.95	5.67X	8.26X
	10p	11	143	2218	6.27	3.95	4.10	4.69	6.77	13.99	39.27	134.19	469.25	1,700.51	15.39	-	1,700.51	6.11X	8.62X
	11p	12	168	2780	10.91	7.46	7.77	9.10	13.51	28.25	77.18	241.64	813.60	2,841.93	10,230.51	36.98	10,230.51	>2.51X	>3.51X
Cars	2p	3	27	216	0.01	0	0	-	-	-	-	-	-	-	-	-	0.01	1.00X	1.00X
	3p	5	60	616	0.83	0.08	0.09	0.10	0.11	-	-	-	-	-	-	-	0.83	463.39X	675.54X
	4p	7	105	1283	75.83	1.33	1.32	1.48	1.47	1.54	1.52	-	-	-	-	-	75.83	>426.09X	>474.74X

Table 11. Experimental Results for Parallel Computation with Segmented Subsumption (Two)

Benchmarks					Time (Reordered-Expansion: Block+: Two)															
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.02	0.01	-	-	-	-	-	-	-	-	-	-	-	-	0.02	0.33X	0.50X
	3p	3	33	-	0.08	0.06	0.06	-	-	-	-	-	-	-	-	-	-	-	0.08	0.85X	2.13X
	4p	4	56	-	0.51	0.56	0.44	0.33	-	-	-	-	-	-	-	-	-	-	0.56	33.05X	108.59X
	5p	5	85	-	7.33	5.63	4.11	2.26	1.53	-	-	-	-	-	-	-	-	-	7.33	356.49X	1,014.51X
	6p	6	120	-	48.73	180.93	302.22	116.35	58.27	14.28	-	-	-	-	-	-	-	-	302.22	>49.95X	>119.12X
	7p	7	161	-	1,009.57	5,800.58	5,944.38	4,729.79	1,418.55	718.13	100.28	-	-	-	-	-	-	-	5,944.38	>1.83X	>6.06X
Scheduler	3p	3	33	336	0.07	0.06	0.06	-	-	-	-	-	-	-	-	-	-	-	0.07	0.89X	2.43X
	4p	4	56	609	0.49	0.48	0.44	0.37	-	-	-	-	-	-	-	-	-	-	0.49	2.34X	8.49X
	5p	5	85	1017	3.16	4.42	3.57	2.44	1.62	-	-	-	-	-	-	-	-	-	4.42	8.93X	30.72X
	6p	6	120	1587	30.50	45.44	70.04	30.00	19.86	5.69	-	-	-	-	-	-	-	-	70.04	37.42X	107.67X
	7p	7	161	2352	409.76	713.09	750.41	466.37	235.03	146.37	31.63	-	-	-	-	-	-	-	750.41	>13.08X	>47.97X
Fischer	6p	7	63	710	0.44	0.36	0.33	0.39	0.45	0.79	0.43	-	-	-	-	-	-	-	0.79	2.88X	11.62X
	7p	8	80	987	1.07	0.74	0.75	0.79	0.97	1.25	2.85	1.03	-	-	-	-	-	-	2.85	6.26X	20.76X
	8p	9	99	1327	1.86	1.43	1.50	1.63	1.78	2.64	3.95	11.14	2.36	-	-	-	-	-	11.14	13.21X	33.54X
	9p	10	120	1736	3.96	2.82	2.88	2.94	3.51	4.25	8.32	13.73	43.96	5.59	-	-	-	-	43.96	25.51X	53.37X
	10p	11	143	2218	6.16	4.96	5.25	5.66	6.16	9.00	12.24	31.03	54.23	187.85	12.93	-	-	-	187.85	43.71X	78.07X
	11p	12	168	2780	11.98	9.25	9.70	10.06	12.43	14.46	28.29	41.72	125.05	218.08	768.56	30.48	-	-	768.56	>28.12X	>46.84X
	12p	13	195	3453	16.99	15.11	18.96	21.41	23.27	36.89	45.97	116.61	175.17	568.44	972.30	3,437.67	72.98	-	3,437.67	>6.52X	>10.47X
	13p	14	224	4397	33.59	46.27	51.49	53.28	67.60	73.81	156.35	187.12	544.83	772.99	2,596.30	4,147.56	14,885.30	195.51	14,885.30	>1.51X	>2.42X
Cars	2p	3	27	216	0.02	0	0.02	-	-	-	-	-	-	-	-	-	-	-	0.02	0.25X	0.50X
	3p	5	60	616	1.04	0.22	0.23	0.25	0.27	-	-	-	-	-	-	-	-	-	1.04	278.96X	539.13X
	4p	7	105	1283	84.46	2.52	2.51	2.74	2.68	2.70	2.74	-	-	-	-	-	-	-	84.46	>358.74X	>426.24X

Table 12. Experimental Results for Parallel Computation with Segmented Subsumption (Three)

Benchmarks					Time (Reordered-Expansion: Block+: Three)															
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0.01	0.01	-	-	-	-	-	-	-	-	-	-	-	-	0.01	0.50X	1.00X
	3p	3	33	-	0.08	0.08	0.08	-	-	-	-	-	-	-	-	-	-	-	0.08	0.71X	2.13X
	4p	4	56	-	0.57	0.61	0.48	0.36	-	-	-	-	-	-	-	-	-	-	0.61	30.10X	99.69X
	5p	5	85	-	4.72	5.10	4.13	2.37	2.37	-	-	-	-	-	-	-	-	-	5.10	397.88X	1458.11X
	6p	6	120	-	70.31	185.54	183.11	144.43	43.51	21.89	-	-	-	-	-	-	-	-	185.54	>55.49X	>194.03X
	7p	7	161	-	1,243.21	6,448.61	11,367.73	4,793.80	1,678.43	759.46	129.51	-	-	-	-	-	-	-	11,367.73	>1.36X	>3.17X
Scheduler	3p	3	33	336	0.07	0.06	0.07	-	-	-	-	-	-	-	-	-	-	-	0.07	0.85X	2.43X
	4p	4	56	609	0.53	0.58	0.51	0.42	-	-	-	-	-	-	-	-	-	-	0.58	2.04X	7.17X
	5p	5	85	1017	10.18	11.10	7.89	4.80	3.76	-	-	-	-	-	-	-	-	-	11.10	3.60X	12.23X
	6p	6	120	1587	39.01	51.71	51.06	40.59	17.51	9.62	-	-	-	-	-	-	-	-	51.71	36.00X	145.84X
	7p	7	161	2352	458.80	800.71	1,337.00	514.26	423.28	173.42	53.08	-	-	-	-	-	-	-	1,337.00	>9.57X	>26.93X
Fischer	6p	7	63	710	0.50	0.38	0.39	0.40	0.48	0.77	0.48	-	-	-	-	-	-	-	0.77	2.70X	11.92X
	7p	8	80	987	1.03	0.82	0.82	0.85	0.98	1.21	1.84	1.23	-	-	-	-	-	-	1.84	6.74X	32.15X
	8p	9	99	1327	1.99	1.56	1.61	1.72	1.87	2.27	4.33	7.52	2.52	-	-	-	-	-	7.52	14.72X	49.68X
	9p	10	120	1736	3.78	3.01	3.05	3.15	3.37	4.71	6.36	9.77	29.07	5.78	-	-	-	-	29.07	32.56X	80.70X
	10p	11	143	2218	6.85	5.34	5.57	5.67	6.59	7.61	9.61	21.21	34.83	66.63	15.74	-	-	-	66.63	78.99X	220.09X
	11p	12	168	2780	11.32	9.16	9.71	10.76	11.79	13.39	24.73	36.63	58.18	191.20	357.30	30.84	-	-	357.30	>47.06X	>100.76X
	12p	13	195	3453	18.51	17.69	19.32	19.96	21.11	33.91	41.13	52.76	160.55	247.41	420.38	1,563.39	73.52	-	1,563.39	>13.38X	>23.03X
	13p	14	224	4397	30.82	30.16	57.67	61.76	79.98	80.45	91.80	230.24	267.03	365.31	1,338.83	2,051.84	3,503.46	240.65	3,503.46	>4.27X	>10.28X
Cars	2p	3	27	216	0.02	0.01	0.01	-	-	-	-	-	-	-	-	-	-	-	0.02	0.25X	0.50X
	3p	5	60	616	0.65	0.24	0.23	0.25	0.25	-	-	-	-	-	-	-	-	-	0.65	343.99X	862.62X
	4p	7	105	1283	74.29	2.39	2.56	2.74	2.66	2.70	2.44	-	-	-	-	-	-	-	74.29	>400.98X	>484.59X

Table 13. Experimental Results for Parallel Computation with Segmented Subsumption (Four)

Benchmarks					Time (Reordered-Expansion: Block+: Four)																
Name		Loc	Dim	Line	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Max	Speedup(original)	Speedup
Scheduler'	2p	2	16	-	0	0.02	-	-	-	-	-	-	-	-	-	-	-	-	0.02	0.50X	0.50X
	3p	3	33	-	0.09	0.07	0.07	-	-	-	-	-	-	-	-	-	-	-	0.09	0.74X	1.89X
	4p	4	56	-	0.74	0.80	0.64	0.51	-	-	-	-	-	-	-	-	-	-	0.80	22.61X	76.01X
	5p	5	85	-	6.76	5.94	6.45	3.01	2.19	-	-	-	-	-	-	-	-	-	6.76	305.39X	1,100.05X
	6p	6	120	-	801.28	609.96	424.12	264.74	190.76	179.55	-	-	-	-	-	-	-	-	801.28	>14.57X	>44.93X
Scheduler	3p	3	33	336	0.05	0.08	0.08	-	-	-	-	-	-	-	-	-	-	-	0.08	0.81X	2.13X
	4p	4	56	609	1.22	0.92	0.77	0.68	-	-	-	-	-	-	-	-	-	-	1.22	1.16X	3.41X
	5p	5	85	1017	19.09	19.72	15.56	10.43	7.80	-	-	-	-	-	-	-	-	-	19.72	1.87X	6.89X
	6p	6	120	1587	218.92	139.75	98.58	59.09	42.02	38.46	-	-	-	-	-	-	-	-	218.92	12.64X	34.45X
	7p	7	161	2352	2,201.23	2,353.41	1,784.56	986.37	777.18	460.28	423.88	-	-	-	-	-	-	-	2,353.41	>4.01X	>15.30X
Fischer	6p	7	63	710	0.67	0.53	0.55	0.56	0.62	0.79	0.68	-	-	-	-	-	-	-	0.79	2.09X	11.62X
	7p	8	80	987	1.31	1.06	1.11	1.09	1.28	1.48	2.15	1.38	-	-	-	-	-	-	2.15	5.44X	27.52X
	8p	9	99	1327	2.61	2.10	2.15	2.20	2.34	2.70	4.47	6.60	3.01	-	-	-	-	-	6.60	13.25X	56.61X
	9p	10	120	1736	5.02	4.00	4.12	4.15	4.68	5.16	6.63	9.54	25.92	6.73	-	-	-	-	25.92	30.89X	90.51X
	10p	11	143	2218	8.91	7.33	7.61	7.69	8.16	8.92	14.16	18.51	31.61	58.66	18.79	-	-	-	58.66	77.04X	249.99X
	11p	12	168	2780	14.23	11.65	12.52	12.55	15.68	16.83	21.23	27.80	73.70	111.71	230.93	34.67	-	-	230.93	>61.70X	>155.89X
	12p	13	195	3453	24.86	22.02	24.78	21.90	25.80	27.70	51.41	63.60	95.44	145.89	538.21	857.67	77.55	-	857.67	>18.21X	>41.97X
	13p	14	224	4397	44.02	40.31	58.40	59.24	78.20	61.62	91.45	105.36	372.57	442.64	706.02	1,125.37	4,868.35	186.11	4,868.35	>4.37X	>7.39X
Cars	2p	3	27	216	0.02	0.01	0.02	-	-	-	-	-	-	-	-	-	-	-	0.02	0.20X	0.50X
	3p	5	60	616	0.79	0.29	0.28	0.33	0.35	-	-	-	-	-	-	-	-	-	0.79	274.85X	709.75X
	4p	7	105	1283	83.52	3.27	3.25	3.34	3.37	3.43	3.41	-	-	-	-	-	-	-	83.52	>347.52X	>431.03X

C AN EXAMPLE OF BENCHMARKS

```
#define time_cnt __attribute__((time_cnt))
#define p1_p1 __attribute__((p1_p1))
#define p2_p2 __attribute__((p2_p2))
#define p3_p3 __attribute__((p3_p3))
#define p4_p4 __attribute__((p4_p4))
#define p5_p5 __attribute__((p5_p5))
#define p6_p6 __attribute__((p6_p6))
#define p7_p7 __attribute__((p7_p7))
#define p0_p2 __attribute__((p0_p2))
#define p0_p1 __attribute__((p0_p1))
#define p1_p2 __attribute__((p1_p2))
#define p2_p1 __attribute__((p2_p1))
#define p2_p0 __attribute__((p2_p0))
#define p1_p0 __attribute__((p1_p0))
#define p1_p3 __attribute__((p1_p3))
#define p3_p1 __attribute__((p3_p1))
#define p1_p4 __attribute__((p1_p4))
#define p4_p1 __attribute__((p4_p1))
#define p1_p5 __attribute__((p1_p5))
#define p5_p1 __attribute__((p5_p1))
#define p1_p6 __attribute__((p1_p6))
#define p6_p1 __attribute__((p6_p1))
#define p1_p7 __attribute__((p1_p7))
#define p7_p1 __attribute__((p7_p1))
#define p2_p3 __attribute__((p2_p3))
#define p3_p2 __attribute__((p3_p2))
#define p2_p4 __attribute__((p2_p4))
#define p4_p2 __attribute__((p4_p2))
#define p2_p5 __attribute__((p2_p5))
#define p5_p2 __attribute__((p5_p2))
#define p2_p6 __attribute__((p2_p6))
#define p6_p2 __attribute__((p6_p2))
#define p2_p7 __attribute__((p2_p7))
#define p7_p2 __attribute__((p7_p2))
#define p3_p4 __attribute__((p3_p4))
#define p4_p3 __attribute__((p4_p3))
#define p3_p5 __attribute__((p3_p5))
#define p5_p3 __attribute__((p5_p3))
#define p3_p6 __attribute__((p3_p6))
#define p6_p3 __attribute__((p6_p3))
#define p3_p7 __attribute__((p3_p7))
#define p7_p3 __attribute__((p7_p3))
```

```
#define p4_p5 __attribute__((p4_p5))
#define p5_p4 __attribute__((p5_p4))
#define p4_p6 __attribute__((p4_p6))
#define p6_p4 __attribute__((p6_p4))
#define p4_p7 __attribute__((p4_p7))
#define p7_p4 __attribute__((p7_p4))
#define p5_p6 __attribute__((p5_p6))
#define p6_p5 __attribute__((p6_p5))
#define p5_p7 __attribute__((p5_p7))
#define p7_p5 __attribute__((p7_p5))
#define p6_p7 __attribute__((p6_p7))
#define p7_p6 __attribute__((p7_p6))

static int x1 = 0;
static int x2 = 0;
static int c1 = 0;
static int k1 = 0;
static int k2 = 0;
static int c2 = 0;
static int time_cnt t = 0;

static void p1_p1 l1evolve(void) {
  if (x1 <= 4) {
    c1 = c1 + t;
    c2 = c2 + t;
    x1 = x1 + t;
    x2 = x2;
    k1 = k1;
    k2 = k2;
  }
}

static void p1_p1 l1increasel1(void) {
  if (c1 >= 10) {
    k1 = k1 + 1;
    c1 = 0;
    c2 = c2;
    x1 = x1;
    x2 = x2;
    k2 = k2;
    t = t;
  }
}
```

```
static void p1_p1 l1decrease(void) {
  if (x1 == 4 && k1 >= 2) {
    k1 = k1 - 1;
    x1 = 0;
    c1 = c1;
    c2 = c2;
    x2 = x2;
    k2 = k2;
    t = t;
  }
}

static void p2_p2 l2evolve(void) {
  if (x2 <= 8) {
    c1 = c1 + t;
    c2 = c2 + t;
    x2 = x2 + t;
    x1 = x1;
    k1 = k1;
    k2 = k2;
  }
}

static void p2_p2 l2increasel1(void) {
  if (c1 >= 10) {
    k1 = k1 + 1;
    c1 = 0;
    c2 = c2;
    x1 = x1;
    x2 = x2;
    k2 = k2;
    t = t;
  }
}

static void p2_p2 l2increasel2(void) {
  if (c2 >= 20) {
    k2 = k2 + 1;
    c2 = 0;
    c1 = c1;
    x1 = x1;
    x2 = x2;
    k1 = k1;
    t = t;
```

```
  }
}

static void p2_p2 l2decrease(void) {
  if (x2 == 8 && k2 >= 2) {
    k2 = k2 - 1;
    x2 = 0;
    c1 = c1;
    c2 = c2;
    x1 = x1;
    k1 = k1;
    t = t;
  }
}

static void p1_p2 l1tol2(void) {
  if (c2 >= 20) {
    c2 = 0;
    k2 = 1;
    c1 = c1;
    x1 = x1;
    x2 = x2;
    k1 = k1;
    t = t;
  }
}

static void p2_p1 l2tol1(void) {
  if (x2 == 8 && k1 >= 1 && k2 == 1) {
    x2 = 0;
    k2 = k2 - 1;
    c1 = c1;
    c2 = c2;
    x1 = x1;
    k1 = k1;
    t = t;
  }
}
```