COMPUTATION TOOLS 2015

The Sixth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking

ISBN: 978-1-61208-394-0

March 22 - 27, 2015

Nice, France

COMPUTATION TOOLS 2015 Editors

Claus-Peter Rückemann, Leibniz Universität Hannover / Westfälische Wilhelms-Universität Münster / North-German Supercomputing Alliance (HLRN), Germany

Pascal Lorenz, University of Haute Alsace, France

COMPUTATION TOOLS 2015

Forward

The Sixth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS 2015), held between March 22-27, 2015 in Nice, France, continued a series of events dealing with logics, algebras, advanced computation techniques, specialized programming languages, and tools for distributed computation. Mainly, the event targeted those aspects supporting context-oriented systems, adaptive systems, service computing, patterns and content-oriented features, temporal and ubiquitous aspects, and many facets of computational benchmarking.

The conference had the following tracks:

- Advanced computation techniques
- Tools for distributed computation
- Logics

Similar to the previous edition, this event attracted excellent contributions and active participation from all over the world. We were very pleased to receive top quality contributions.

We take here the opportunity to warmly thank all the members of the COMPUTATION TOOLS 2015 technical program committee, as well as the numerous reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and effort to contribute to COMPUTATION TOOLS 2015. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the COMPUTATION TOOLS 2015 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope COMPUTATION TOOLS 2015 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of computational logics, algebras, programming, tools, and benchmarking. We also hope that Nice, France provided a pleasant environment during the conference and everyone saved some time to enjoy the charm of the city.

COMPUTATION TOOLS 2015

Committee

COMPUTATION TOOLS Advisory Chairs

Kenneth Scerri, University of Malta, Malta
Alexander Gegov, University of Portsmouth, UK
Ahmed Khedr, University of Sharjah, UAE

COMPUTATIONAL TOOLS Industry/Research Chairs

Torsten Ullrich, Fraunhofer Austria Research GmbH - Graz, Austria
Zhiming Liu, UNU-IIST, Macao

COMPUTATION TOOLS Publicity Chair

Lev Naiman, University of Toronto, Canada
Ingram Bondin, University of Malta, Malta
Tomáš Bublík, Czech Technical University in Prague, Czech Republic

COMPUTATION TOOLS 2015 Technical Program Committee

Yas Aksultanny, Arabian Gulf University, Bahrain
Youssif B. Al-Nashif, Old Dominion University, USA
Adel Alimi, University of Sfax, Tunisia
François Anton, Technical University of Denmark, Denmark
Henri Basson, University of Lille North of France (Littoral), France
Steffen Bernhard, TU-Dortmund, Germany
Ateet Bhalla, Oriental Institute of Science & Technology - Bhopal, India
Paul-Antoine Bisgambiglia, Université de Corse, France
Narhimene Boustia, Saad Dahlab University - Blida, Algeria
Azahara Camacho-Magriñán, Universidad de Cádiz, Spain
Luca Cassano, University of Pisa, Italy
Emanuele Covino, Università di Bari, Italy
Hepu Deng, RMIT University - Melbourne, Australia
Rene de Souza Pinto, University of Sao Paulo, Brazil
Craig C. Douglas, University of Wyoming / Yale University, USA
António Dourado, University of Coimbra, Portugal
Eugene Feinberg, Stony Brook University, USA
Tommaso Flaminio, University of Insubria, Italy

Copyright Information

Table of Contents

Advanced Computation of a Sparse Precision Matrix
HADAP: A Hadamard-Dantzig Estimation of a Sparse Precision Matrix

Mohammed Elanbari*, Reda Rawi†, Michele Ceccarelli†, Othmane Bouhali‡, Halima Bensmail†
*Sidra Medical and Research Center, Qatar Foundation
†Qatar Computing Research Institute, Qatar Foundation
‡Texas A&M University-Qatar,
Doha, Qatar

Email: *melanbari@sidra.org; †{rrawi, mceccarelli, hbensmail}@qf.org.qa; ‡othmane.bouhali@qatar.tamu.edu

***Abstract*—Estimating large sparse precision matrices is an interesting and challenging problem in many fields of sciences, engineering, and humanities, thanks to advances in computing technologies. Recent applications often encounter high dimensionality with a limited number of data points leading to a number of covariance parameters that greatly exceeds the number of observations. Several methods have been proposed to deal with this problem, but there is no guarantee that the obtained estimator is positive definite. Furthermore, in many cases, one needs to capture some additional information on the setting of the problem. In this work, we propose an innovative approach named HADAP for estimating the precision matrix by minimizing a criterion combining a relaxation of the gradient-log likelihood and a penalization of lasso type. We derive an efficient Alternating Direction Method of multipliers algorithm to obtain the optimal solution.**

Keywords–Covariance matrix; Frobenius norm; Gaussian graphical model; Precision matrix; Alternating method of multipliers; Positive-definite estimation; Sparsity..

I. INTRODUCTION

Recent applications often encounter high dimensionality with a limited number of data points leading to a number of covariance parameters that greatly exceeds the number of observations. Examples include marketing, e-commerce, and warehouse data in business; microarray, and proteomics data in genomics and heath sciences; and biomedical imaging, functional magnetic resonance imaging, tomography, signal processing, high-resolution imaging, and functional and longitudinal data. In biological sciences, one may want to classify diseases and predict clinical outcomes using microarray gene expression or proteomics data, in which hundreds of thousands of expression levels are potential covariates, but there are typically only tens or hundreds of subjects. Hundreds of thousands of single-nucleotide polymorphisms are potential predictors in genome-wide association studies. The dimensionality of the variables spaces grows rapidly when interactions of such predictors are considered. Large-scale data analysis is also a common feature of many problems in machine learning, such as text and document classification and computer vision. For a $p \times p$ covariance matrix Σ, there are $p(p+1)/2$ parameters to estimate, yet the sample size n is often small. In addition, the positive-definiteness of Σ makes the problem even more complicated. When $n > p$, the sample covariance matrix is positive-definite and unbiased, but as the dimension p increases, the sample covariance matrix tends to become unstable and can fail to be consistent.

II. BACKGROUND

In this paper, we use the notation in Table I.

TABLE I. NOTATION USED IN THIS PAPER

Notation	Description
$\boldsymbol{A} \succeq 0$	$\boldsymbol{A} \in \mathbb{R}^{n\times p}$ is symmetric and positive semidefinite
$\boldsymbol{A} \succ 0$	$\boldsymbol{A} \in \mathbb{R}^{n\times p}$ is symmetric and positive definite
$\lVert\boldsymbol{A}\rVert_1$	ℓ_1 norm of $\boldsymbol{A} \in \mathbb{R}^{n\times p}$, i.e $\sum_{ij} a_{ij}$
$\lVert\boldsymbol{A}\rVert_\infty$	ℓ_∞ norm of $\boldsymbol{A} \in \mathbb{R}^{n\times p}$, i.e $max_{ij} a_{ij}$
$\lVert\boldsymbol{A}\rVert_2$	spectral norm of $\boldsymbol{A} \in \mathbb{R}^{i\times j}$ i.e. the maxmum eigenvalues of $\boldsymbol{A} \succ 0$
$\lVert\boldsymbol{A}\rVert_F$	Frobenius norm of $\boldsymbol{A} \in \mathbb{R}^{n\times p}$ i.e $\sqrt{\sum_{ij} a_{ij}^2}$
$\mathrm{Tr}(\boldsymbol{A})$	trace of $\boldsymbol{A} \in \mathbb{R}^{p\times p}$, i.e $\mathrm{Tr}(\boldsymbol{A}) = \sum_i a_{ii}$
vec$(\mathbf{A})$	stacked form of $\boldsymbol{A} \in \mathbb{R}^{n\times p}$, i.e vec$(\boldsymbol{A}) = (a_{1,1}, \dots, a_{n,1}, a_{1,2}, \dots, a_{1,p}, \dots, a_{n,p})^t$,
vec$(\boldsymbol{ABC})$	$(\boldsymbol{C}^t \otimes \boldsymbol{A})$ vec$(\boldsymbol{B})$
vec$(\boldsymbol{A} \circ \boldsymbol{B})$	diag(vec$(\boldsymbol{A})$)vec$(\boldsymbol{B})$
diag(A)	$\mathrm{diag}(\boldsymbol{u}) = \begin{pmatrix} u_1 & 0 & \cdots & 0 \\ 0 & u_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_k \end{pmatrix} \in \mathbb{R}^{N\times N}$,
$\boldsymbol{A} \circ \boldsymbol{B}$	Hadamard product of $\boldsymbol{A}$ and $\boldsymbol{B}$, $\in \mathbb{R}^{N\times M}$, i.e element-wise multiplication $(\boldsymbol{A} \circ \boldsymbol{B})_{ij} = (\boldsymbol{A})_{ij} \times (\boldsymbol{B})_{ij}$
$\boldsymbol{A} \otimes \boldsymbol{B}$	Kronecker product of $\boldsymbol{A}$ and $\boldsymbol{B}$ $\boldsymbol{A} \otimes \boldsymbol{B} = \begin{pmatrix} a_{1,1}\boldsymbol{B} & a_{1,2}\boldsymbol{B} & \cdots & a_{1,m}\boldsymbol{B} \\ a_{2,1}\boldsymbol{B} & a_{2,2}\boldsymbol{B} & \cdots & a_{2,m}\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}\boldsymbol{B} & a_{n,2}\boldsymbol{B} & \cdots & a_{n,m}\boldsymbol{B} \end{pmatrix}$ $\boldsymbol{A} \otimes \boldsymbol{B} \in \mathbb{R}^{np\times mq}$

a) Existing Methods: Due in part to its importance, there has been an active line of work on efficient optimization methods for solving the ℓ_1 regularized Gaussian MLE problem: PSM that uses projected subgradients [1], ALM using alternating linearization [2], IPM an inexact interior point method [11], SINCO a greedy coordinate descent method [3] and Glass a block coordinate descent method [4][5] etc. For typical high-dimensional statistical problems, optimization methods typically suffer sub-linear rates of convergence [6]. This would be too expensive for the Gaussian MLE problem, since the number of matrix entries scales quadratically with the number of nodes.

b) Sparse Modeling: Sparse modeling has been widely used to deal with high dimensionality. The main assumption is that the p-dimensional parameter vector is sparse, with many components being exactly zero or negligibly small. Such an assumption is crucial in identifiability, especially for the relatively small sample size. Although the notion of sparsity gives rise to biased estimation in general, it has proved to be effective in many applications. In particular, variable selection can increase the estimation accuracy by effectively identifying important predictors and can improve the model interpretability. To solve this, constraints are frequently imposed on the covariance to reduce the number of parameters in the model including the spectral decomposition, Bayesian methods, modeling the matrix-logarithm, nonparametric smoothing, banding/thresholding techniques (see [5], [7][8]).

Specifically, thresholding is proposed for estimating permutation-invariant consistent covariance matrices when the true covariance matrix is bandable [4]. In this sense, thresholding is more robust than banding/tapering for real applications. In this paper we focus on the soft-thresholding technique as in [4] and [9] because it can be formulated as the solution of a convex optimization problem. In fact, Graphical Lasso approach (Glasso) is introduced as the following. Let $\|.\|_F$ be the Frobenius norm and $|.|_1$ be the element-wise ℓ_1-norm of all non-diagonal elements. Then the soft-thresholding covariance estimator is equal to

$$\hat{\Omega}^{+} = \arg\min_{\Omega} \frac{1}{2}\|\Omega - \hat{\Omega}_n\|_F^2 + \lambda|\Omega|_1, \tag{1}$$

where $\Omega = \Sigma^{-1}$, where $\Omega = \omega_{ij}{}_{1\leq i,j\leq p}$ is the precision matrix, $\hat{\Omega}$ is the solution of (1) and λ is a tuning parameter. This equation emphasizes the fact that the solution Ω may not be unique [such nonuniqueness can occur if rank(Ω) $< p$]. It has been shown that the existence of a robust optimization formulation is related to kernel density estimation [10] where property of the solution and a proof that Lasso is consistent was given using robustness directly. Moreover, [11] proved that there exists a linear subspace that is almost surely unique, meaning that it will be the same under different boundary sets corresponding to different solutions of equations of type (1). However, there is no guarantee that the thresholding estimator is always positive definite (see [9], [12] and [13]). Although the positive definite property is guaranteed in the asymptotic setting with high probability, the actual estimator can be an indefinite matrix, especially in real data analysis.

Structure of the inverse covariance matrix has attracted special interest. Example, when dealing with asset allocation in finance. The financial problems are often written in terms of the inverse covariance matrix, in such case the covariance structure of return series is often estimated using only the most recent data, resulting in a small sample size compared to the number of parameters to be estimated. In biological applications (graphical models), zero correlations represent conditional independence between the variables. For example to account for network information in the analysis of metabolites data, the reconstruction of metabolic networks from a collection of metabolite patterns is a key question in the computational research field. Previous attempts focused on linear metabolite associations measured by Pearson correlation coefficients [14]. A major drawback of correlation networks, however, is their inability to distinguish between direct and indirect associations. Correlation coefficients are generally high in large-scale omics data sets, suggesting a plethora of indirect and systemic associations. Gaussian Graphical models (GGMs) circumvent indirect association effects by evaluating conditional dependencies in multivariate Gaussian distributions or equivalently the inverse covariance matrix (see [15]).
On the other hand, additional structure on the precision matrix coefficients is also often required (Comparative genomic hybridizaton) where the difference between two successive coefficients is required to be small or to vary slowly.

Our Contributions: In this paper, we emphasize on introducing a new criteria that insures the positive-definiteness of the covariance matrix adding a tuning parameter $\epsilon > 0$ in the constraints. This additional constraint will guard against positive semi-definite. We add structure on the coefficient of the precision matrix and we derive an efficient ADMM algorithm form to obtain an optimal solution. We perform Alternating Direction Method of Multipliers steps, a variant of the standard Augmented Lagrangian method, that uses partial updates, but with three innovations that enable finessing the caveats detailed above.

In Section III, we link the gaussian graphical model to the precision matrix estimation, in which we show that recovering the structure of a graph G is equivalent to the estimation of the support of the precision matrix and describe different penalized optimization algorithm that have been used to solve this problem and their limitations.

We describe, in Section IV, the ADMM and its application to solve the estimation of the precision matrix under the Dantzig-selector setting and we show its ability to perform distributed optimization where we break up the big optimization problem into smaller problems that are more manageable. In fact, as in the recent methods [9][12], we build on the observation that the Newton direction computation is a Lasso problem, and perform iterative coordinate descent to solve this Lasso problem. Then, we use a Dantzig-selector rule to obtain a step-size that ensures positive-definiteness of the next iterate. In Section V, we validate our algorithm on artificial and real data.

III. Link with Gaussian graphical model (GGM)

Given a data set consisting of samples from a zero mean Gaussian distribution in $\mathbb{R}^p$,

$$X^{(i)} \sim \mathcal{N}(\mathbf{0}, \Sigma), i = 1, ..., n, \tag{2}$$

with positive definite $p \times p$ covariance matrix Σ. Note that the zero mean assumption in equation 2 is mainly for simplifying notation. The task here is how to estimate the precision matrix $\Omega = \Sigma^{-1}$ when it is sparse. We are particularly interested by the case of sparse Ω because it is closely linked to the selection of graphical models.

To be more specific, let $G = (V, E)$ be a graph representing conditional independence relations between components of $\mathbf{X} = (X_1, ..., X_p)$.

- The vertex set V has p components $X_1, ..., X_p$;
- The edge set E consists of ordered pairs (i, j) of $V \times V$, where $(i, j) \in E$ if there is an edge between X_i and X_j.

We exclude the edge between two vertexes X_i and X_j if and only if X_i and X_j are independent given $\{X_k, k \neq i, j\}$. If in addition $\mathbf{X} \sim \mathcal{N}(\mathbf{0}, \Sigma)$, the conditional independence between X_i and X_j given the remaining variables is equivalent to $\omega_{ij} = 0$, where $\Omega = \Sigma^{-1} = \{\omega_{ij}\}_{1 \le i,j \le p}$. Hence, in the case of Gaussian Graphical Model (GGM), the edges are given by the inverse of covariance matrix. More precisely, recovering the structure of a graph G is equivalent to the estimation of the support of the precision matrix Ω. When $n > p$, one can estimate Σ^{-1} by maximum likelihood, but when $p > n$ this is not possible because the empirical covariance matrix is singular and often performs poorly [16]. Since the ℓ_1-norm is the tighest convex upper bound of the cardinality of a matrix, several ℓ_1-regularization methods have been proposed. Consequently a number of authors have considered minimizing an ℓ_1 penalized log-likelihood (see [4] and [17]). It is also sometimes called Glasso [4] after the algorithm that efficiently computes the solution. It consists of solving the penalized problem

$$\hat{\Omega}_{\text{Glasso}} = \arg\min_{\Omega \succeq \mathbf{0}} \{\langle \hat{\Sigma}_n, \Omega\rangle - \log\det\Omega + \lambda\|\Omega\|_1\}. \quad (3)$$

where $\hat{\Sigma}_n$ is the sample covariance matrix and λ is a tuning parameter. The term $\|\Omega\|_1$ encourages sparseness of the precision matrix. The asymptotic properties of the estimator has been studied in [17]. Lauritzen [13] proposed a constrained l1 minimization proposed a constrained l_1 minimization procedure for estimating sparse precision matrices by solving the optimization problem

$$\text{minimize} \quad \|\Omega\|_1 \quad \text{subject to} \quad \|\hat{\Sigma}_n\Omega - \mathbf{I}\|_\infty \le \lambda, \quad (4)$$

where λ is a tuning parameter. The authors established the rates of convergence under both the entry-wise l_∞ and the Frobenius norm. In computationally viewpoint, equation 4 can be solved by remarking that it can be decomposed into a series of Dantzig selector problems [18]. This observation is useful in both implementation and technical analysis. Theoretically, the authors prove that the estimator is positive definite with high probability. However, in practice there is no guarantee that the estimator is always positive definite, especially in real data analysis.

IV. Alternating Direction Method of Multipliers for HADAP

We define our algorithm HADAP as a solution to the following problem

$$\hat{\Omega}^+ = \arg\min_{\Omega \succeq \epsilon\mathbf{I}} \frac{1}{2}\|\hat{\Sigma}_n\Omega - \mathbf{I}\|_F^2 + \lambda|\mathbf{D}\circ\Omega|_1, \quad (5)$$

where $\hat{\Sigma}_n$ is the empirical covariance matrix, $\mathbf{D} = (d_{ij})_{1\le i,j\le p}$ is an arbitrary matrix with non-negative elements where $\mathbf{D}$ can take different forms: it can be the matrix of all ones or it can be a matrix with zeros on the diagonal to avoid shrinking diagonal elements of Ω. Furthermore, we can take $\mathbf{D}$ with elements $d_{ij} = 1_{i\neq j}|\hat{\Omega}_{ij}^{\text{init}}|$, where $\hat{\Omega}^{\text{init}}$ is an initial estimator of Ω. The later choice of $\mathbf{D}$ corresponds to precision matrix analogue of the Adaptive Lasso penalty.

We propose an ADMM algorithm to solve the problem (5). To derive ADMM, we will first introduce a new variable Θ and an equality constraint as follows:

$$\begin{aligned}(\hat{\Theta}^+, \hat{\Omega}^+) &= \arg\min_{\Theta,\Omega} \left\{\frac{1}{2}\|\hat{\Sigma}_n\Omega - \mathbf{I}\|_F^2 + \lambda|\mathbf{D}\circ\Omega|_1 \right.\\ &: \quad \left. \Omega = \Theta, \Theta \succeq \epsilon\mathbf{I}\right\}. \quad (6)\end{aligned}$$

The solution to (6) gives the solution to (5). To deal with the problem (6), we have to minimize its augmented Lagrangian function for some given penalty parameter ρ, i.e.

$$\begin{aligned}L_\rho(\Theta,\Omega,\Lambda) &= \frac{1}{2}\|\hat{\Sigma}_n\Omega - \mathbf{I}\|_F^2 + \lambda|\mathbf{D}\circ\Omega|_1 - \langle\Lambda, \Theta - \Omega\rangle \\ &+ \frac{1}{2\rho}\|\Theta - \Omega\|_F^2, \quad (7)\end{aligned}$$

where Λ is the Lagrange multiplier.

At iteration k, the ADMM algorithm consists of the three steps, namely Θ-Step, Ω-Step and the dual-update step :

$$\Theta^{k+1} := \arg\min_{\Theta\succeq\epsilon\mathbf{I}} L_\rho(\Theta, \Omega^k, \Lambda^k); \quad \Theta\text{-minimization} \quad (8)$$

$$\Omega^{k+1} := \arg\min_{\Omega} L_\rho(\Theta^{k+1}, \Omega, \Lambda^k); \quad \Omega\text{-minimization} \quad (9)$$

$$\Lambda^{k+1} := \Lambda^k - \frac{1}{\rho}(\Theta^{k+1} - \Omega^{k+1}). \quad \text{dual-update} \quad (10)$$

- In the first step of the ADMM algorithm, we fix Ω and Λ and minimize the augmented Lagrangian over Θ.
- In the second step, we fix Θ and Λ and minimize the augmented Lagrangian over Ω.
- Finally, we update the dual variable Λ.

To further simplify the ADMM algorithm, we will derive the closed-form solutions for (8)-(10).

A. The Θ-Step.

Let $\mathbf{A}^+$ be the projection of a matrix $\mathbf{A}$ onto the convex cone $\{\Theta \succeq \epsilon\mathbf{I}\}$. Assume that $\mathbf{A}$ has the eigen-decomposition $\sum_{j=1}^p \lambda_j \mathbf{v}_j\mathbf{v}_j^t$. Then, it is well known that $\mathbf{A}^+ = \sum_{j=1}^p \max(\epsilon,\lambda_j)\mathbf{v}_j\mathbf{v}_j^t$. Using this property, the Θ-Step can be analytically solved as follows

$$\begin{aligned}\Theta^{k+1} &= \arg\min_{\Theta\succeq\epsilon\mathbf{I}} L_\rho(\Theta,\Omega^k,\Lambda^k) \\ &= \arg\min_{\Theta\succeq\epsilon\mathbf{I}} \frac{1}{2}\|\hat{\Sigma}_n\Omega^k - \mathbf{I}\|_F^2 + \lambda|\mathbf{D}\circ\Omega^k|_1 \\ &- \langle\Lambda^k, \Theta - \Omega^k\rangle + \frac{1}{2\rho}\|\Theta - \Omega^k\|_F^2 \\ &= \arg\min_{\Theta\succeq\epsilon\mathbf{I}} -\langle\Lambda^k,\Theta\rangle + \frac{1}{2\rho}\|\Theta - \Omega^k\|_F^2 \\ &= \arg\min_{\Theta\succeq\epsilon\mathbf{I}} \|\Theta - (\Omega^k + \rho\Lambda^k)\|_F^2 \\ &= (\Omega^k + \rho\Lambda^k)^+ \\ &= \sum_{j=1}^p \max(\epsilon,\lambda_j)\mathbf{v}_j\mathbf{v}_j^t,\end{aligned}$$

where $\sum_{j=1}^p \lambda_j\mathbf{v}_j\mathbf{v}_j^t$ is eigen-decomposition of $\Omega^k + \rho\Lambda^k$.

B. The Ω-Step.

$$\begin{aligned}\Omega^{k+1} &= \arg\min_{\Omega} L_\rho(\Theta^{k+1},\Omega,\Lambda^k)\\ &= \arg\min_{\Omega} \frac{1}{2}\|\hat\Sigma_n\Omega-\mathbf{I}\|_F^2+\lambda|\mathbf{D}\circ\Omega|_1\\ &- \langle\Lambda^k,\Theta^{k+1}-\Omega\rangle+\frac{1}{2\rho}\|\Theta^{k+1}-\Omega\|_F^2\\ &= \arg\min_{\Omega} \frac{1}{2}\|\hat\Sigma_n\Omega-\mathbf{I}\|_F^2+\lambda|\mathbf{D}\circ\Omega|_1\\ &+ \langle\Lambda^k,\Omega\rangle+\frac{1}{2\rho}\|\Theta^{k+1}-\Omega\|_F^2\\ &= \arg\min_{\Omega} \frac{1}{2}\|\hat\Sigma_n\Omega-\mathbf{I}\|_F^2+\frac{1}{2\rho}\|\Omega-\Theta^{k+1}\|_F^2\\ &+ \langle\Lambda^k,\Omega\rangle+\lambda|\mathbf{D}\circ\Omega|_1.\end{aligned}$$

We have

$$\begin{aligned}\frac{1}{2}\|\hat\Sigma_n\Omega-\mathbf{I}\|_F^2 &= \frac{1}{2}\langle\hat\Sigma_n\Omega-\mathbf{I},\hat\Sigma_n\Omega-\mathbf{I}\rangle\\ &= \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2-2\langle\hat\Sigma_n\Omega,\mathbf{I}\rangle+\|\mathbf{I}\|_F^2\right\}\\ &= \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2-2\operatorname{Tr}(\Omega^t\hat\Sigma_n^t\mathbf{I})+\operatorname{Tr}(\mathbf{I}^t\mathbf{I})\right\}\\ &= \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2-2\operatorname{Tr}(\Omega^t\hat\Sigma_n)+p\right\}\\ &= \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2-2\langle\Omega,\hat\Sigma_n\rangle+p\right\}.\end{aligned}$$

and

$$\begin{aligned}\frac{1}{2\rho}\|\Omega-\Theta^{k+1}\|_F^2 &= \frac{1}{2\rho}\langle\Omega-\Theta^{k+1},\Omega-\Theta^{k+1}\rangle\\ &= \frac{1}{2\rho}\left\{\|\Omega\|_F^2-2\langle\Omega,\Theta^{k+1}\rangle+\|\Theta^{k+1}\|_F^2\right\}.\end{aligned}$$

Then, the Ω-Step is equivalent to

$$\begin{aligned}\Omega^{k+1} &= \arg\min_{\Omega} \frac{1}{2}\|\hat\Sigma_n\Omega-\mathbf{I}\|_F^2+\frac{1}{2\rho}\|\Omega-\Theta^{k+1}\|_F^2\\ &+ \langle\Lambda^k,\Omega\rangle+\lambda|\mathbf{D}\circ\Omega|_1\\ &= \arg\min_{\Omega} \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2-2\langle\Omega,\hat\Sigma_n\rangle+p\right\}\\ &+ \frac{1}{2\rho}\left\{\|\Omega\|_F^2-2\langle\Omega,\Theta^{k+1}\rangle+\|\Theta^{k+1}\|_F^2\right\}+\langle\Lambda^k,\Omega\rangle\\ &+ \lambda|\mathbf{D}\circ\Omega|_1\\ &= \arg\min_{\Omega} \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2-2\langle\Omega,\hat\Sigma_n\rangle\right\}\\ &+ \frac{1}{2\rho}\left\{\|\Omega\|_F^2-2\langle\Omega,\Theta^{k+1}\rangle\right\}\\ &+ \langle\Lambda^k,\Omega\rangle+\lambda|\mathbf{D}\circ\Omega|_1\\ &= \arg\min_{\Omega} \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2+\frac{1}{\rho}\|\Omega\|_F^2\right.\\ &- \left.2\langle\Omega,\hat\Sigma_n+\frac{1}{\rho}\Theta^{k+1}-\Lambda^k\rangle\right\}+\lambda|\mathbf{D}\circ\Omega|_1\\ &= \arg\min_{\Omega} \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2+\frac{1}{\rho}\|\Omega\|_F^2\right.\\ &- \left.\frac{2}{\rho}\langle\Omega,\rho(\hat\Sigma_n-\Lambda^k)+\Theta^{k+1}\rangle\right\}+\lambda|\mathbf{D}\circ\Omega|_1.\end{aligned}$$

At this level, we are not able to derive a closed form for Ω. To overcome this problem, we propose to derive a new ADMM to update Ω. To do this, we reparametrize the $\mathbf{D}\circ\Omega$ with Γ and we add an equality constraint $\mathbf{D}\circ\Omega=\Gamma$, then we minimize

$$\frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2+\frac{1}{\rho}\|\Omega\|_F^2-\frac{2}{\rho}\langle\Omega,\rho(\hat\Sigma_n-\Lambda^k)+\Theta^{k+1}\rangle\right\}+\lambda|\Gamma|_1$$

subject to

$$\mathbf{D}\circ\Omega=\Gamma. \tag{11}$$

The augmented Lagrangian associated to this problem $L_\rho^k(\Omega,\Gamma,\Delta)$ is

$$\begin{aligned}&\frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2+\frac{1}{\rho}\|\Omega\|_F^2-\frac{2}{\rho}\langle\Omega,\rho(\hat\Sigma_n-\Lambda^k)+\Theta^{k+1}\rangle\right\}\\ +\ &\lambda|\Gamma|_1-\langle\Delta,\Gamma-\mathbf{D}\circ\Omega\rangle+\frac{1}{2\rho}\|\Gamma-\mathbf{D}\circ\Omega\|_F^2,\end{aligned}\tag{12}$$

where Δ is the Lagrange multiplier and ρ is the same parameter as in (7).
As before, the ADMM for this problem consists of the following three intermediate steps:

$$\Omega_k^{j+1} := \arg\min_{\Omega} L_\rho^k(\Omega,\Gamma^j,\Delta^j);\quad \Omega\text{-minimization} \tag{13}$$

$$\Gamma^{j+1} := \arg\min_{\Gamma} L_\rho^k(\Omega_k^{j+1},\Gamma,\Delta^j);\quad \Gamma\text{-minimization} \tag{14}$$

$$\Delta^{j+1} := \Delta^j-\frac{1}{\rho}(\Omega_k^{j+1}-\Gamma^{j+1}).\quad \text{dual-update} \tag{15}$$

C. The intermediate Ω-Step.

$$\begin{aligned}\Omega_k^{j+1} &= \arg\min_{\Omega} L_\rho^k(\Omega,\Gamma^j,\Delta^j)\\ &= \arg\min_{\Omega}\frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2+\frac{1}{\rho}\|\Omega\|_F^2\right.\\ &- \left.\frac{2}{\rho}\langle\Omega,\rho(\hat\Sigma_n-\Lambda^k)+\Theta^{k+1}\rangle\right\}+\lambda|\Gamma^j|_1\\ &- \langle\Delta^j,\Gamma^j-\mathbf{D}\circ\Omega\rangle+\frac{1}{2\rho}\|\Gamma^j-\mathbf{D}\circ\Omega\|_F^2\\ &= \frac{1}{2}\left\{\|\hat\Sigma_n\Omega\|_F^2+\frac{1}{\rho}\|\Omega\|_F^2-\frac{2}{\rho}\langle\Omega,\rho(\hat\Sigma_n-\Lambda^k)\right.\\ &+ \left.\Theta^{k+1}\rangle\right\}+\langle\Delta^j,\mathbf{D}\circ\Omega\rangle+\frac{1}{2\rho}\|\Gamma^j-\mathbf{D}\circ\Omega\|_F^2\end{aligned}$$

$$=\frac{1}{2}\operatorname{Tr}\left\{\Omega^t\hat\Sigma_n^t\hat\Sigma_n\Omega+\frac{1}{\rho}\Omega^t\Omega-\frac{2}{\rho}\Omega^t(\rho(\hat\Sigma_n-\Lambda^k)+\Theta^{k+1})\right\}$$

$$+\operatorname{Tr}((\Delta^j)^t\mathbf{D}\circ\Omega)+\frac{1}{2\rho}\operatorname{Tr}((\Gamma^j-\mathbf{D}\circ\Omega)^t(\Gamma^j-\mathbf{D}\circ\Omega)).$$

Then, the partial differential of L_ρ^k with respect to Ω, $\frac{\partial L_\rho^k(\Omega,\Gamma^j,\Delta^j)}{\partial\Omega}$ is

$$=\frac{1}{2}\left\{2\hat\Sigma_n^t\hat\Sigma_n\Omega+\frac{2}{\rho}\Omega-\frac{2}{\rho}(\rho(\hat\Sigma_n-\Lambda^k)+\Theta^{k+1})\right\}$$

$$+\Delta^j\circ\mathbf{D}+\frac{1}{2\rho}\left\{-2\Gamma^j\circ\mathbf{D}+2\mathbf{D}\circ\mathbf{D}\circ\Omega\right\}$$

$$=\left(\hat\Sigma_n^t\hat\Sigma_n+\frac{1}{\rho}\mathbf{I}\right)\Omega+\frac{1}{\rho}\mathbf{D}\circ\mathbf{D}\circ\Omega$$

$$+\left(\Delta^j - \frac{1}{\rho}\Gamma^j\right)\circ \mathbf{D} - \frac{1}{\rho}\left(\rho(\hat{\Sigma}_n - \Lambda^k) + \Theta^{k+1}\right).$$

Since Ω_k^{j+1} is the minimizer of $L_\rho^k(.,\Gamma^j,\Delta^j)$, we must have

$$\frac{\partial L_\rho^k(\Omega_k^{j+1},\Gamma^j,\Delta^j)}{\partial\Omega} = \mathbf{0},$$

which is equivalent to

$$\left(\hat{\Sigma}_n^t\hat{\Sigma}_n + \frac{1}{\rho}\mathbf{I}\right)\Omega_k^{j+1} + \frac{1}{\rho}\mathbf{D}\circ\mathbf{D}\circ\Omega_k^{j+1}$$

$$+\left(\Delta^j - \frac{1}{\rho}\Gamma^j\right)\circ\mathbf{D} - \frac{1}{\rho}\left(\rho(\hat{\Sigma}_n - \Lambda^k) + \Theta^{k+1}\right) = \mathbf{0}.$$

Finally, Ω_k^{j+1} has a closed form given by the previous expression despite the additional but straightforward computational effort at this level. This additional step, can be considered as a warming start for the original ADMM algorithm. This is very important when dealing with complex problem and large data.

D. The intermediate Γ-Step.

To deal with this Step, define an entry-wise soft-thresholding rule for all the off-diagonal elements of a matrix $\mathbf{A}$ as $\mathbf{S}(\mathbf{A},\kappa) = \{s(a_{jl},\kappa)\}_{1\le j,l\le p}$ with

$$s(a_{jl},\kappa) = \text{sign}(a_{jl})\max(|a_{jl}| - \kappa, 0)I_{\{j\neq l\}}.$$

Then the Γ-Step has a closed form given by

$$\begin{aligned}
\Gamma^{j+1} &= \arg\min_{\Gamma} L_\rho^k(\Omega_k^{j+1},\Gamma,\Delta^j)\\
&= \arg\min_{\Gamma}\frac{1}{2}\left\{\|\hat{\Sigma}_n\Omega_k^{j+1}\|_F^2 + \frac{1}{\rho}\|\Omega_k^{j+1}\|_F^2\right.\\
&- \left.\frac{2}{\rho}\langle\Omega_k^{j+1}, \rho(\hat{\Sigma}_n - \Lambda^k) + \Theta^{k+1}\rangle\right\}\\
&+ \lambda|\Gamma|_1 - \langle\Delta^j, \Gamma - \mathbf{D}\circ\Omega_k^{j+1}\rangle + \frac{1}{2\rho}\|\Gamma - \mathbf{D}\circ\Omega_k^{j+1}\|_F^2\\
&= \arg\min_{\Gamma}\frac{1}{2}\|\Gamma - \mathbf{D}\circ\Omega_k^{j+1}\|_F^2 - \rho\langle\Delta^j,\Gamma\rangle + \rho\lambda|\Gamma|_1\\
&= \arg\min_{\Gamma}\frac{1}{2}\left\{\|\Gamma\|_F^2 - 2\langle\rho\Delta^j + \mathbf{D}\circ\Omega_k^{j+1},\Gamma\rangle\right\}\\
&+ \rho\lambda|\Gamma|_1\\
&= \arg\min_{\Gamma}\frac{1}{2}\|\Gamma - (\rho\Delta^j + \mathbf{D}\circ\Omega_k^{j+1})\|_F^2 + \rho\lambda|\Gamma|_1\\
&= \mathbf{S}(\rho\Delta^j + \mathbf{D}\circ\Omega_k^{j+1}, \rho\lambda).
\end{aligned}$$

Algorithm 1 shows complete details of HADAP using ADMM (see Figure 1).

```
Algorithm 1: HADAP algorithm
initialize the variables: $\Theta^0 = \mathbf{0}, \Omega^0 = \mathbf{0}, \Lambda^0 = \mathbf{0}$,
$\Gamma^0 = \mathbf{0}$, $\Delta^0 = \mathbf{0}$ ;
Select a scalar $\rho > 0$;
for $k = 0, 1, 2, ...$ until convergence do
    $\Theta^{k+1} := (\Omega^k + \rho\Lambda^k)^+$;
    for $j = 0, 1, 2, ...$ until convergence do
        $A_\Sigma^{j+1} \leftarrow \left(\rho\hat{\Sigma}_n^t\hat{\Sigma}_n + \mathbf{D}^t\mathbf{D} + \mathbf{I}\right)^{-1}$ ;
        $B_\Sigma^{j+1} \leftarrow$
        $\left((1-\rho)\mathbf{D}^t\Delta^j + \rho(\hat{\Sigma}_n - \Lambda^k) + \Theta^{k+1}\right)$ ;
        $\Omega_k^{j+1} := A_\Sigma \times B_\Sigma$ ;
        $\Gamma^{j+1} := \mathbf{S}(\rho\Delta^j + \mathbf{D}\Omega_k^{j+1}, \rho\lambda)$;
        $\Delta^{j+1} := \Delta^j - \frac{1}{\rho}(\Omega_k^{j+1} - \Gamma^{j+1})$;
    end
    $\Omega^{k+1} := \lim_{j\to+\infty}\Omega_k^j$;
    $\Lambda^{k+1} := \Lambda^k - \frac{1}{\rho}(\Theta^{k+1} - \Omega^{k+1})$;
end
```

Figure 1. Complete details of HADAP using the Alternating Direction Method of Multipliers.

V. EXPERIMENTS

Among existing methods, the lasso penalized Gaussian likelihood estimator is the only popular matrix precision estimator that can simultaneously retain sparsity and positive definiteness. To show the goodness of our approach, we use simulations and real example to compare the performance of our estimator with Glasso.

A. Validation on synthetic data

In order to validate our approach, we used the same simulation structure as in [13]. We generated n = 1000 samples from a p = 600-dimensional normal distribution with correlation structure of the form $\sigma(x_i,x_j) = 0.6|i-j|$. This model has a banded structure, and the values of the entries decay exponentially as they move away from the diagonal. We generated an independent sample of size 1000 from the same distribution for validating the tuning parameter λ. Using the training data, we compute a series of estimators with 50 different values of λ and use the one with the smallest likelihood loss on the validation sample, where the likelihood loss [19], is defined by

$$L(\Sigma,\Omega) = \langle\Sigma,\Omega\rangle - \log det(\Omega) \qquad (16)$$

We mention that all the experiments are conducted on a PC with 4 Gb RAM, 3Ghz CPU using Matlab 2009a.

B. Measurable quantities and results.

Output displays the primal residual $|r^k|$, the primal feasibility tolerance epsilon ϵ^{pri}, the dual residual s^k, and the dual feasibility tolerance ϵ^{dual} quantities. Also included is a plot of the objective values by iterations. Note that the objective value at any particular iteration can go below the true solution value p^* because the iterates does not need to be feasible (e.g., if the constraint is $x - z = 0$, we can have $x^k - z^k \neq 0$ for some k).
Results for $\lambda = 0.01$ are summarized in Figure 2 and Table II. The convergence is achieved in 25 steps and need just 0.54 seconds. After a few steps of fluctuations ($\approx$ 12 iterations), the objective function stabilizes and converges to its optimal value where the eigenvalues of the precision matrix estimated by the HADAP are real and positive, which prove the positive definiteness of the obtained precision matrix as shown in Figure 3.

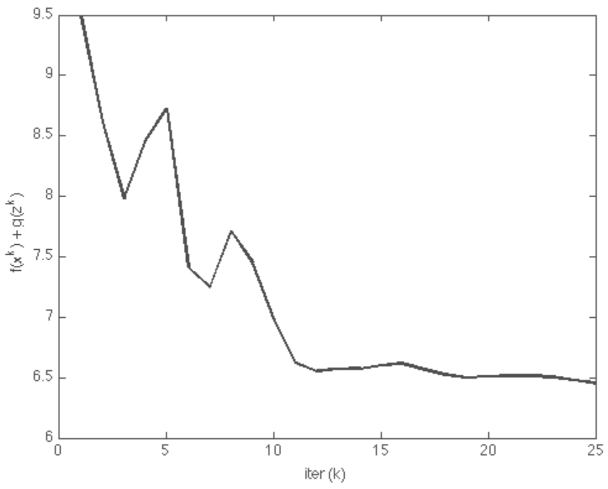


Figure 2. Plot of the objective function for $\lambda = 0.01$

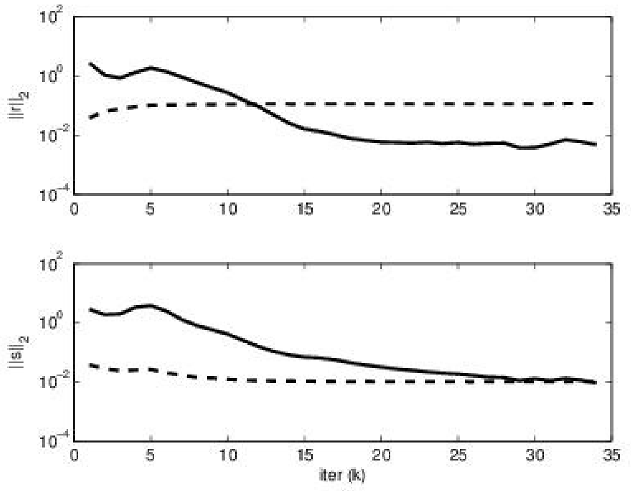


Figure 3. Plot of the objective function

TABLE II. VALUES OF THE PRIMAL RESIDUAL $|r^k|$, THE PRIMAL FEASIBILITY TOLERANCE ϵ^{pri}, THE DUAL RESIDUAL s^k, AND THE DUAL FEASIBILITY TOLERANCE ϵ^{dual} for $\lambda = 0.01$. ALSO INCLUDED THE OBJECTIVE VALUES BY ITERATION.

iter	r.norm	eps.pri	s.norm	eps.dual	objective
1	2.76	0.04	2.86	0.04	8.49
2	1.08	0.07	1.86	0.03	8.11
3	0.88	0.08	1.99	0.02	8.89
4	1.34	0.09	3.37	0.03	7.95
5	1.88	0.10	3.77	0.03	8.64
6	1.42	0.11	2.52	0.02	8.94
7	0.94	0.11	1.28	0.02	7.79
8	0.62	0.11	0.81	0.01	9.70
9	0.41	0.11	0.58	0.01	7.70
10	0.28	0.11	0.42	0.01	6.70
11	0.16	0.11	0.25	0.01	6.71
12	0.09	0.11	0.16	0.01	6.73
13	0.05	0.11	0.11	0.01	6.75
14	0.03	0.11	0.08	0.01	6.77
15	0.02	0.11	0.07	0.01	6.79
16	0.01	0.12	0.06	0.01	6.80
17	0.01	0.12	0.06	0.01	6.82
18	0.01	0.12	0.04	0.01	6.83
19	0.01	0.12	0.04	0.01	6.84
20	0.01	0.12	0.03	0.01	6.85
21	0.01	0.12	0.03	0.01	6.86
22	0.01	0.12	0.02	0.01	6.86
23	0.01	0.12	0.02	0.01	6.87
24	0.01	0.12	0.02	0.01	6.88
25	0.01	0.12	0.02	0.01	6.88

C. Validation on real data

For experimental validation, we used 4 cancer datasets publicly available at the Gene Expression Omnibus [20]. For a fair comparison with the other method of estimating the inverse covariance matrix, we follow the same analysis scheme used by [19]. Datasets are: Liver cancer (GSE1898), Colon cancer (GSE29638), Breast cancer (GSE20194) and Prostate cancer (GSE17951) with sample size $n = 182; 50; 278$ and 154 respectively and number of genes $p = 21794; 22011; 22283$ and 54675. We preprocessed the data so that each variable is zero mean and unit variance across the dataset. We performed 100 repetitions on a $50\%-50\%$ validation and testing samples.

Since regular sparseness promoting methods do not scale to large number of variables, we used the same regime proposed by [19] and validated our method in two regimes. In the first regime, for each of the 50 repetitions, we selected $n = 200$ variables uniformly at random and use the glasso. In the second regime, we use all the variables in the dataset, and use the method dpglasso from [21]. Since the whole sample covariance matrix could not fit in memory, we computed it in batches of rows [21]. In order to make a fair comparison, the runtime includes the time needed to produce the optimal precision matrix from a given input dataset. Average runtimes was summarized in table III. This includes not only the time to solve each optimization problem but also the time to compute the covariance matrix (if needed). Our HADAP method is considerably faster than the Glasso method as shown in table III .

TABLE III. RUNTIMES FOR GENE EXPRESSION DATASETS. OUR HADAP METHOD IS CONSIDERABLY FASTER THAN SPARSE METHOD.

Dataset	*Graphical lasso*	*Our estimator*
GSE1898	3.8 min	1.0 min
GSE29638	3.8 min	2.6 min
GSE20194	3.8 min	2.5 min
GSE17951	14.9min	4.8 min

VI. Conclusion and Future Work

The sparse precision matrix estimator has been shown to be useful in many applications. Penalizing the matrix is a tool with good asymptotic properties for estimating large sparse covariance and precision matrices. However, its positive definiteness property and unconstrained structure can be easily violated in practice, which prevents its use in many important applications such as graphical models, financial assets and comparative genomic hybridization. In this paper, we have expressed the precision matrix estimation equation in a convex optimization framework and considered a natural modification by imposing the positive definiteness and problem-solving constraints. We have developed a fast alternating direction method to solve the constrained optimization problem and the resulting estimator retains the sparsity and positive definiteness properties simultaneously. We are at the phase of demonstrating the general validity of the method and its advantages over correlation networks based on competitive precision matrix estimators with computer-simulated reaction systems, to be able to demonstrate strong signatures of intracellular pathways and provide a valuable tool for the unbiased reconstruction of metabolic reactions from large-scale metabolomics data sets.

REFERENCES

[1] J. Duchi, S. Gould, and D. Koller, "Projected Subgradient Methods for Learning Sparse Gaussians," in Proceedings of the Twenty-fourth Conference on Uncertainty in AI (UAI), 2008.

[2] K. Scheinberg, S. Ma, and D. Goldfarb, "Sparse inverse covariance selection via alternating linearization methods," in Advances in Neural Information Processing, NIPS10, 2010.

[3] L. Li and K. Toh, "An inexact interior point method for l1-reguarlized sparse covariance selection," Mathematical Programming Computation, vol. 2, 2010, pp. 291–315.

[4] J. Friedman, T. Hastie, and R. Tibshirani, "Sparse Inverse Covariance Estimation With the Graphical Lasso," Biostatistics, vol. 9, 2008, pp. 432–441.

[5] O. Banerjee, L. El Ghaoui, and A. d'Aspremont, "Model selection through sparse maximum likelihood estimation for multivariate Gaussian or binary data," Journal of Machine Learning Research, vol. 9, 2008, pp. 485–516.

[6] A. Agarwal, S. Negahban, and M. Wainwright, "Convergence rates of gradient methods for high-dimensional statistical recovery," in Advances in Neural Information Processing, NIPS10, 2010.

[7] P. Bickel and E. Levina, "Regularized estimation of large covariance matrices," Ann. Statist., vol. 36, 2008a, pp. 199–227.

[8] R. Yang and J. Berger, "Estimation of a covariance matrix using the reference prior," Ann. Statist., vol. 3, 1994, pp. 1195–1211.

[9] X. Lingzhou, S. Ma, and H. Zhou, "Positive Definite L_1 Penalized Estimation of Large Covariance Matrices," Journal of the American Statistical Association, vol. 500, 2012, pp. 1480–1491.

[10] H. Xu, C. Caramanis, and S. Mannor, "Robust Regression and Lasso," IEEE Transaction on Information Theory, vol. 56, 2010, pp. 3561–357.

[11] R. Tibshirani and J. Taylor, "The solution path of the generalized lasso," Ann. Statist, vol. 39 (3), 2011, pp. 1335–1371.

[12] N. El Karoui, "Operator norm consistent estimation of large dimensional sparse covariance matrices," Annals of Statistics, vol. 36, 2008, pp. 2717–2756.

[13] T. Cai and H. Zhou, "A constrained l_1 minimization approach to sparse precision matrix estimation," J. American Statistical Association, vol. 106, 2011, pp. 594–607.

[14] J. Krumsiek, K. Suhre, T. Illig, J. Adamski, and F. Theis, "Gaussian graphical modeling reconstructs pathway reactions from high-throughput metabolomics data," BMC Systems Biology, vol. 5 (21), 2011, pp. 2–16.

[15] S. Lauritzen, Graphical Models. Clarendon Press, Oxford, 1996.

[16] I. Johnstone, "On the distribution of the largest eigenvalue in principal components analysis," Ann. Statist., vol. 29 (2), 2001, pp. 295–327.

[17] P. Ravikumar, W. M.J., G. Raskutti, and B. Yu, "High-dimensional covariance estimation by minimizing l_1-penalized log-determinant divergence," Electron. J. Statist., vol. 5, 2011, pp. 935–980.

[18] E. Candes and T. Tao, "The Dantzig selector: statistical estimation when p is much larger than n," Annals of Statistics, vol. 35, 2007, pp. 2313–2351.

[19] J. Honorio and T. Jaakkola, "Inverse covariance estimation for high-dimensional data in linear time and space: Spectral methods for riccati and sparse models," in Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence, 2013.

[20] R. Edgar, M. Domrachev, and A. Lash, "Gene Expression Omnibus: NCBI gene expression and hybridization array data repository," Nucleic Acids Res, vol. 30 (1), 2002, pp. 207–210.

[21] R. Mazumder and T. Hastie, "Exact covariance thresholding into connected components for largescale graphical lasso," The Journal of Machine Learning Research, vol. 13, 2012, pp. 781–794.

A Specialized Recursive Language for Capturing Time-Space Complexity Classes

Emanuele Covino and Giovanni Pani
Dipartimento di Informatica
Università di Bari, Italy
Email: emanuele.covino@uniba.it, giovanni.pani@uniba.it

***Abstract*—We provide a resource-free characterization of register machines that computes their output within polynomial time $O(n^k)$, by defining our version of *predicative recursion* and a related recursive programming language. Then, by means of some restriction on composition of programs, we define a programming language that characterize the register machines with a polynomial bound imposed over time and space complexity, simultaneously. A simple syntactical analysis allows us to evaluate the complexity of a program written in these languages.**

***Keywords*–Specialized computation languages, time-space classes, implicit computational complexity, predicative recursion.**

I. Introduction

The definition of a complexity class is usually made by imposing an explicit bound on time and/or space resources used by a Turing Machine (or another equivalent model) during its computation. On the other hand, different approaches use logic and formal methods to provide languages for complexity-bounded computation; they aim at studying computational complexity without referring to external measuring conditions or a particular machine model, but only by considering language restrictions or logical/computational principles implying complexity properties. In particular, this is achieved by characterizing complexity classes by means of recursive operators with explicit syntactical restrictions on the role of variables.

The first characterization of this type was given by Cobham [4], in which the polynomial-time computable functions are exactly those functions generated by bounded recursion on notation; Leivan [8] and Bellantoni and Cook [1] gave other characterizations of PTIMEF. Several other complexity classes have been characterized by means of unlimited operators: see, for instance, Leivant and Marion [9] and Oitavem [10] for PSPACEF and the class of the elementary functions; Clote [3] for the definition of a time/space hierarchy between PTIMEF and PSPACEF; Leivant [6], [7] and [8] for a theoretical insight. All these approaches have been dubbed *Implicit Computational Complexity*: they share the idea that no explicitly bounded schemes are needed to characterize a great number of classes of functions and that, in order to do this, it suffices to distinguish between *safe* and *unsafe* variables (or, following Simmons [11], between *dormant* and *normal* ones) in the recursion schemes. This distinction yields many forms of *predicative* recursion, in which the being-defined function cannot be used as counter into the defining one. The two main objectives of this area are to find natural implicit characterizations of various complexity classes of functions, thereby illuminating their nature and importance, and to design methods suitable for static verification of program complexity. This approach represent a bridge between the complexity theory and the programming language theory; a mere syntactical inspection allows us to evaluate the complexity of a given program written in one of the previous mentioned specialized languages.

Our version of the *safe recursion* scheme on a binary word algebra is such that $f(x, y, za) = h(f(x, y, z), y, za)$; throughout this paper we will call x, y and z the auxiliary variable, the parameter, and the principal variable of a program defined by recursion, respectively. We don't allow the renaming of variable z as x, and this implies that the step program h cannot assign the previous value of the being-defined program f to the principal variable z: in other words, we always know in advance the number of recursive calls of the step program in a recursive definition. We obtain that z is a *dormant* variable, according to Simmons' approach, or a *safe* one, following Bellantoni and Cook.

In Section II, starting from a natural definition of constructors and destructors over an algebra of lists, we give our definition of recursion-free programs and of the safe recursion scheme. In section III, we recall the definition of computation by register machines as provided by [8]. In section IV, we define the hierarchy of classes of programs $\mathcal{T}_k$, with $k \in \mathbb{N}$, where programs in $\mathcal{T}_1$ can be computed by register machines within linear time, and $\mathcal{T}_{k+1}$ are programs obtained by one application of safe recursion to elements in $\mathcal{T}_k$; we prove that they are computable within time $O(n^k)$. We then restrict $\mathcal{T}_k$ to the hierarchy $\mathcal{S}_k$, whose elements are the programs computable by a register machine in linear space. By means of a restricted form of composition between programs, we define, in Section V, a polytime-space hierarchy $\mathcal{TS}_{qp}$, such that each program in $\mathcal{TS}_{qp}$ can be computed by a register machine within time $O(n^p)$ and space $O(n^q)$, simultaneously. We have that $\bigcup_{k<\omega} \mathcal{T}_k$ captures PTIME. Even though this is a well-known result (see [8]), we use it to prove the second one (see also [5] and [9] for other approaches to the characterization of joint time-space classes). Both results are a preliminary step for an implicit classification of the hierarchy of time-space classes between PTIME and PSPACE, as defined in [3]. In Section VI we summarize the results and give some hints about future work.

II. BASIC INSTRUCTIONS AND DEFINITION SCHEMES

In this section we introduce the basic operators of our programming language (for a different approach see [2]). The language is defined over a binary word algebra, with the only restriction that words are packed into lists, with the symbol © acting as a separator between words. In this way, we are able to handle a sequence of words as a single object.

A. Recursion-free programs and classes $\mathcal{T}_0$ and $\mathcal{S}_0$

$\mathbf{B}$ is the binary alphabet $\{0,1\}$. $a, b, a_1, \ldots$ denotes elements of $\mathbf{B}$, and $U, V, \ldots, Y$ denotes words over $\mathbf{B}$. $p, q, \ldots, s, \ldots$ denotes lists in the form $Y_1 © Y_2 © \ldots © Y_{n-1} © Y_n$. ϵ is the empty word. The *i-th component* $(s)_i$ of $s = Y_1 0 Y_2 0 \ldots 0 Y_{n-1} 0 Y_n$ is Y_i. $|s|$ is the length of the list s, that is the number of letters occurring in s. We write x, y, z for the variables used in a program, and we write u for one among x, y, z. Programs will be denoted with f, g, h, and they will have the form $f(x, y, z)$, where some among the variables may be absent.

Definition 2.1: The *basic instructions* are:

1) the *identity* $\text{I}(u)$, that returns the value s assigned to u;
2) the *constructors* $\text{C}_i^a(s)$, that adds the digit a at the right of the last digit of $(s)_i$, with $a = 0, 1$ and $i \geq 1$;
3) the *destructors* $\text{D}_i(s)$, that erases the rightmost digit of $(s)_i$, with $a = 0, 1$ and $i \geq 1$.

Constructors $\text{C}_i^a(s)$ and destructors $\text{D}_i(s)$ leave s unchanged if it has less than i components.

Example 2.1: Given the word $s = 01©11©©00$, we have that $|s| = 9$ and $(s)_2 = 11$. We also have $\text{C}_1^1(01©11) = 011©11$, $\text{D}_2(0©0©) = 0©©$, $\text{D}_2(0©©) = 0©©$.

Definition 2.2: Given the programs g and h, f is defined by *simple schemes* if it is obtained by:

1) *renaming* of x as y in g, that is, f is the result of the substitution of the value of y to all occurrences of x into g. Notation: $f = \text{RNM}_{x/y}(g)$;
2) *renaming* of z as y in g, that is, f is the result of the substitution of the value of y to all occurrences of z into g. Notation: $f = \text{RNM}_{z/y}(g)$;
3) *selection* in g and h, when for all s, t, r we have

$$f(s,t,r) = \begin{cases} g(s,t,r) & \text{if the rightmost digit} \\ & \text{of } (s)_i \text{ is } b \\ h(s,t,r) & \text{otherwise,} \end{cases}$$

with $i \geq 1$ and $b = 0, 1$. Notation: $f = \text{SEL}_i^b(g, h)$.

Example 2.2: if f is defined by $\text{RNM}_{x/y}(g)$ we have that $f(t, r) = g(t, t, r)$. Similarly, f defined by $\text{RNM}_{z/y}(g)$ implies that $f(s, t) = g(s, t, t)$. Let s be the word $00©1010$, and $f = \text{SEL}_2^0(g, h)$; we have that $f(s, t, r) = g(s, t, r)$, since the rightmost digit of $(s)_2$ is 0.

Definition 2.3: f is obtained by *safe composition* of h and g in the variable u if it is obtained by substitution of h to u in g; if $u = z$, then x must be absent in h. Notation: $f = \text{SCMP}_u(h, g)$.

Definition 2.4: A *modifier* is obtained by the safe composition of a sequence of constructors and a sequence of destructors.

Definition 2.5: $\mathcal{T}_0$ is the class of programs defined by closure of modifiers under SEL and SCMP.

Definition 2.6: Given $f \in \mathcal{T}_0$, the *rate of growth* $rog(f)$ is such that

1) if f is a modifier, $rog(f)$ is the difference between the number of constructors and the number of destructors occurring in its definition;
2) if $f = \text{SEL}_i^b(g, h)$, then $rog(f)$ is $\max(rog(g), rog(h))$;
3) if $f = \text{SCMP}_u(h, g)$, then $rog(f)$ is $\max(rog(g), rog(h))$.

Definition 2.7: $\mathcal{S}_0$ is the class of programs in $\mathcal{T}_0$ with non-positive rate of growth, that is $\mathcal{S}_0 = \{f \in \mathcal{T}_0 | rog(f) \leq 0\}$.

Note that all elements in $\mathcal{T}_0$ and in $\mathcal{S}_0$ modify their inputs according to the result of some test performed over a fixed number of digits. Moreover, elements in $\mathcal{S}_0$ cannot return values longer than their input.

B. Safe recursion and classes $\mathcal{T}_1$ and $\mathcal{S}_1$

Definition 2.8: Given the programs $g(x, y)$ and $h(x, y, z)$, $f(x, y, z)$ is defined by *safe recursion* in the *basis* g and in the *step* h if for all s, t, r we have

$$\begin{cases} f(s,t,a) &= g(s,t) \\ f(s,t,ra) &= h(f(s,t,r),t,ra). \end{cases}$$

Notation: $f = \text{SREC}(g, h)$.

In particular, $f(x, z)$ is defined by *iteration* of $h(x)$ if for all s, r we have

$$\begin{cases} f(s,a) &= s \\ f(s,ra) &= h(f(s,r)). \end{cases}$$

Notation: $f = \text{ITER}(h)$. We write $h^{|r|}(s)$ for $\text{ITER}(h)(s, r)$ (i.e., the $|r|$-*th* iteration of h on s).

Definition 2.9: $\mathcal{T}_1$ (respectively, $\mathcal{S}_1$) is the class defined by closure under simple schemes and SCMP of programs in $\mathcal{T}_0$ (resp., $\mathcal{S}_0$) and programs obtained by one application of ITER to $\mathcal{T}_0$ (resp., $\mathcal{S}_0$).
Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, ITER($\mathcal{T}_0$); SCMP, SIMPLE) (resp., $\mathcal{S}_1$=($\mathcal{S}_0$, ITER($\mathcal{S}_0$); SCMP, SIMPLE)).

As we have already stated in the Introduction, we call x, y and z the auxiliary variable, the parameter, and the principal variable of a program obtained by means of the previous recursion scheme. The renaming of z as x is not allowed (see definitions 2.2 and 2.3), implying that the step program of a recursive definition cannot assign the recursive call to the principal variable. This is the key of the time-complexity bound intrinsic into our programs, together with the limitations imposed to the renaming of variables.

Definition 2.10:
1) Given $f \in \mathcal{T}_1$, the *number of components* of f is $max\{i | \mathrm{D}_i$ or C_i^a or SEL_i^b occurs in $f\}$. Notation: $\#(f)$.
2) Given a program f, its *length* is the number of constructors, destructors and defining schemes occurring in its definition. Notation: $lh(f)$.

III. Computation by register machines

In this section, we recall the definition of register machine (see [8]), and we give the definition of computation within a given time (or space) bound.

Definition 3.1: Given a free algebra $\mathbf{A}$ generated from constructors $\mathbf{c_1}, \ldots, \mathbf{c_n}$ (with $arity(\mathbf{c_i}) = r_i$), a *register machine* over $\mathbf{A}$ is a computational device M having the following components:

1) a finite set of *states* $S = \{s_0, \ldots, s_n\}$;
2) a finite set of *registers* $\Phi = \{\pi_0, \ldots, \pi_m\}$;
3) a collection of *commands*, where a command may be:
   a **branching** $s_i \pi_j s_{i_1} \ldots s_{i_k}$, such that when M is in the state s_i, switches to state $s_{i_1}, \ldots, s_{i_k}$ according to whether the main constructor (i.e., the leftmost) of the term stored in register π_j is $\mathbf{c}_1, \ldots, \mathbf{c}_k$;
   a **constructor** $s_i \pi_{j_1} \ldots \pi_{j_{r_i}} \mathbf{c}_i \pi_l s_r$, such that when M is in the state s_i, store in π_l the result of the application of the constructor $\mathbf{c}_i$ to the values stored in $\pi_{j_1} \ldots \pi_{j_{r_i}}$, and switches to s_r;
   a **p-destructor** $s_i \pi_j \pi_l s_r$ ($p \leq max(r_i)_{i=1\ldots k}$), such that when M is in the state s_i, store in π_l the p-th subterm of the term in π_j, if it exists; otherwise, store the term in π_j. Then it switched to s_r.

A *configuration* of M is a pair (s, F), where $s \in S$ and $F : \Phi \to \mathbf{A}$. M induces a transition relation $\vdash_M$ on configurations, where $\kappa \vdash_M \kappa'$ holds if there is a command of M whose execution converts the configuration κ in κ'. A *computation* of M on input $\vec{X} = X_1, \ldots, X_p$ with output $\vec{Y} = Y_1, \ldots, Y_q$ is a sequence of configurations, starting with (s_0, F_0), and ending with (s_1, F_1) such that:

1) $F_0(\pi_{j'(i)}) = X_i$, for $1 \leq i \leq p$ and j' a permutation of the p registers;
2) $F_1(\pi_{j''(i)}) = Y_i$, for $1 \leq i \leq q$ and j'' a permutation of the q registers;
3) each configuration is related to its successor by $\vdash_M$;
4) the last configuration has no successor by $\vdash_M$.

Definition 3.2: A register machine M *computes* the program f if, for all s, t, r, we have that $f(s, t, r) = q$ implies that M computes $(q)_1, \ldots, (q)_{\#(f)}$ on input $(s)_1, \ldots, (s)_{\#(f)}, (t)_1, \ldots, (t)_{\#(f)}, (r)_1, \ldots, (r)_{\#(f)}$.

Definition 3.3:
1) For each input $\vec{X}$ (with $|\vec{X}| = n$), M computes its output within time $O(p(n))$ if its computation runs through $O(p(n))$ configurations; M computes its output in space $O(q(n))$ if, during the whole computation, the global length of the contents of its registers is $O(q(n))$.
2) For each input $\vec{X}$ (with $|\vec{X}| = n$), M needs time $O(p(n))$ *and* space $O(q(n))$ if the two bounds occur simultaneously, during the same computation.

Note that the number of registers needed by M to compute a given f has to be fixed a priori (otherwise, we should have to define a family of register machines for each program to be computed, with each element of the family associated to an input of a given length). According to definition 2.10 and 3.2, M uses a number of registers which linearly depends on the highest component's index that f can manipulate or access with one of its constructors, destructors or selections; and which depends on the number of times a variable is used by f, that is, on the total number of different copies of the registers that M needs during the computation. Both these numbers are constant values.
Unlike the usual operators *cons*, *head* and *tail* over Lisp-like lists, our constructors and destructors can have direct access to any component of a list, according to definition 2.1. Hence, their computation by means of a register machine requires constant time, but it requires an amount of time which is linear in the length of the input, when performed by a Turing machine.
Codes. We write $s_i © F_j(\pi_0) © \ldots © F_j(\pi_k)$ for the word that encodes a configuration (s_i, F_j) of M, where each component is a binary word over $\{0, 1\}$.

Lemma 3.1: f belongs to $\mathcal{T}_1$ if and only if f is computable by a register machine within time $O(n)$.

Proof: To prove the first implication we show (by induction on the structure of f) that each $f \in \mathcal{T}_1$ can be computed by a register machine M_f in time cn, where c is a constant which depends on the construction of f, and n is the length of the input.

Base. $f \in \mathcal{T}_0$. This means that f is obtained by closure of a number of modifiers under selection and simple schemes; each modifier g can be computed within time bounded by $lh(g)$, the overall number of basic instructions and definition schemes of g, i.e. by a machine running over a constant number of configurations; the result follows, since the safe composition and the selection can be simulated by our model of computation.
Step. Case 1. $f = \mathrm{ITER}(g)$, with $g \in \mathcal{T}_0$. We have that $f(s, r) = g^{|r|}(s)$. A register machine M_f can be defined as follows: $(s)_i$ is stored in the register π_i ($i = 1 \ldots \#(f)$) and $(r)_j$ is stored in the register π_j ($j = \#(f) + 1 \ldots 2\#(f)$); M_f runs M_g (within time $lh(g)$) for $|r|$ times. Each time M_g is called, M_f deletes one digit from one of the registers $\pi_{\#(f)+1} \ldots \pi_{2\#(f)}$, starting from the first; the computation stops, returning the final result, when they are all empty. Thus, M_f computes $f(s, r)$ within time $|r| lh(g)$.
Case 2. Let f be defined by simple schemes or SCMP. The result follows by direct simulation of the schemes.

In order to prove the second implication, we show that the behaviour of a k-register machine M which operates in time cn can be simulated by a program in $\mathcal{T}_1$. Let nxt_M be a program in T_0, such that nxt_M operates on input $s = s_i \copyright F_j(\pi_0) \copyright \ldots \copyright F_j(\pi_k)$ and it has the form $if\ state[i](s)\ then\ E_i$, where $state[i](s)$ is a test which is true if the state of M is s_i, and E_i is a modifier which updates the code of the state and the code of one among the registers, according to the definition of M. By means of $c-1$ SCMP's we define nxt^c_M in $\mathcal{T}_0$, which applies c times nxt_M to the word that encodes a configuration of M. We define in $\mathcal{T}_1$

$$\begin{cases} linsim_M(x, a) = & x \\ linsim_M(x, za) = & nxt^c_M(linsim_M(x, z)) \end{cases}$$

$linsim_M(s, t)$ iterates $nxt_M(s)$ for $c|t|$ times, returning the code of the configuration which contains the final result of M. ∎

IV. THE TIME HIERARCHY

In this section, we recall the definition of classes of programs $\mathcal{T}_1$ and $\mathcal{S}_1$; we define our hierarchy of classes of programs, and we state the relation with the classes of register machines, which compute their output within a polynomially-bounded amount of time.

Definition 4.1: ITER($\mathcal{T}_0$) denotes the class of programs obtained by one application of iteration to programs in $\mathcal{T}_0$. $\mathcal{T}_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{T}_0$ and programs in ITER($\mathcal{T}_0$).
$\mathcal{T}_{k+1}$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{T}_k$ and programs in SREC($\mathcal{T}_k$).
Notation: $\mathcal{T}_1$=($\mathcal{T}_0$, ITER($\mathcal{T}_0$); SCMP, SIMPLE).
$\mathcal{T}_{k+1}$=($\mathcal{T}_k$, SREC($\mathcal{T}_k$); SCMP, SIMPLE).

Definition 4.2: ITER($\mathcal{S}_0$) denotes the class of programs obtained by one application of iteration to programs in $\mathcal{S}_0$. $\mathcal{S}_1$ is the class of programs obtained by closure under safe composition and simple schemes of programs in $\mathcal{S}_0$ and programs in ITER($\mathcal{S}_0$).
$\mathcal{S}_{k+1}$ is the class of programs obtained by closure under simple schemes of programs in $\mathcal{S}_k$ and programs in SREC($\mathcal{S}_k$).
Notation: $\mathcal{S}_1$=($\mathcal{S}_0$, ITER($\mathcal{S}_0$); SCMP, SIMPLE).
$\mathcal{S}_{k+1}$=($\mathcal{S}_k$, SREC($\mathcal{S}_k$); SIMPLE).

Hence, hierarchy $\mathcal{S}_k$, with $k \in \mathbb{N}$, is a version of $\mathcal{T}_k$ in which each program returns a result whose length is *exactly* bounded by the length of the input; this doesn't happen if we allow the closure of $\mathcal{S}_k$ under SCMP. We will use this result to evaluate the space complexity of our programs.

Lemma 4.1: Each $f(s, t, r)$ in $\mathcal{T}_k$ ($k \geq 1$) can be computed by a register machine within time $|s|+lh(f)(|t|+|r|)^k$.

Proof: Base. $f \in \mathcal{T}_1$. The relevant case is when f is in the form ITER(h), with h a modifier in $\mathcal{T}_0$. In lemma 3.1 (case 1 of the step) we have proved that $f(s, r)$ can be computed within time $|r|lh(h)$; hence, we have the thesis.

Step. $f \in \mathcal{T}_{p+1}$. The most significant case is when $f =$SREC(g, h). The inductive hypothesis gives two register machines M_g and M_h which compute g and h within the required time. Let r be the word $a_1 \ldots a_{|r|}$; recalling that $f(s, t, ra) = h(f(s, t, r), t, ra)$, we define a register machine M_f such that it calls M_g on input s, t, and calls M_h for $|r|$ times on input stored into the appropriate set of registers (in particular, the result of the previous recursive step has to be stored always in the same register). By inductive hypothesis, M_g needs time $|s|+lh(g)(|t|)^p$ in order to compute g; for the first computation of the step program h, M_h needs time $|g(s, t)| + lh(h)(|t| + |a_{|r|-1}a_{|r|}|)^p$.
After $|r|$ calls of M_h, the final configuration is obtained within overall time $|s| + \max(lh(g), lh(h))(|t| + |r|)^{p+1} \leq |s| + lh(f)(|t| + |r|)^{p+1}$. ∎

Lemma 4.2: The behaviour of a register machine which computes its output within time $O(n^k)$ can be simulated by an f in $\mathcal{T}_k$.

Proof: Let M be a register machine respecting the hypothesis. As we have already seen, there exists $nxt_M \in \mathcal{T}_0$ such that, for input the code of a configuration of M, it returns the code of the configuration induced by the relation $\vdash_M$. Given a fixed i, we write the program σ_i by means of i safe recursions nested over nxt_M, such that it iterates nxt_M on input s for n^i times, with n the length of the input:

$\sigma_0 :=$ITER(nxt_M) and
$\sigma_{n+1} :=$IDT$_{z/y}(\gamma_{n+1})$, where $\gamma_{n+1} :=$SREC(σ_n, σ_n).

We have that
$\sigma_0(s, t) = nxt^{|t|}_M(s)$, $\sigma_{n+1}(s, t) = \gamma_{n+1}(s, t, t)$, and

$$\begin{cases} \gamma_{n+1}(s, t, a) & = \sigma_n(s, t) \\ \gamma_{n+1}(s, t, ra) & = \sigma_n(\gamma_{n+1}(s, t, r), t) \\ & = \gamma_n(\gamma_{n+1}(s, t, r), t, t) \end{cases}$$

In particular we have

$$\sigma_1(s, t) = \quad \gamma_1(s, t, t) = \quad \underbrace{\sigma_0(\sigma_0(\ldots \sigma_0(s, t) \ldots))}_{|t|\ times} = nxt^{|t|^2}_M$$

$$\sigma_2(s, t) = \quad \gamma_2(s, t, t) = \quad \underbrace{\sigma_1(\sigma_1(\ldots \sigma_1(s, t) \ldots))}_{|t|\ times} = nxt^{|t|^3}_M$$

By induction we see that σ_{k-1} iterates nxt_M on input s for $|t|^k$ times, and that it belongs to $\mathcal{T}_k$. The result follows defining $f(t) = \sigma_{k-1}(t, t)$, with t the code of an initial configuration of M. ∎

Theorem 4.1: f belongs to $\mathcal{T}_k$ if and only if f is computable by a register machine within time $O(n^k)$.

Proof: By lemma 4.1 and lemma 4.2. ∎

We recall that register machines are polytime reducible to Turing machines; this implies that $\bigcup_{k<\omega} \mathcal{T}_k$ captures PTIME (see [8]).

V. The time-space hierarchy

In this section, we define a time-space hierarchy of recursive programs (see [4]), and we state the equivalence with the classes of register machines which compute their output with a simultaneous bound on time and space.

Definition 5.1: Given the programs g and h, f is obtained by *weak composition* of h in g if $f(x,y,z) = g(h(x,y,z),y,z)$. Notation: $f = \text{WCMP}(h,g)$.

In the *weak* form of composition the program h can be substituted only in the variable x, while in the *safe* composition the substitution is possible in all variables.

Definition 5.2: For all $p,q \geq 1$, $\mathcal{TS}_{qp}$ is the class of programs obtained by weak composition of h in g, with $h \in \mathcal{T}_q$, $g \in \mathcal{S}_p$ and $q \leq p$.

Lemma 5.1: For all f in $\mathcal{S}_p$, we have $|f(s,t,r)| \leq \max(|s|,|t|,|r|)$.

Proof: By induction on p. Base. The relevant case is when $f \in \mathcal{S}_1$ and f is defined by iteration of g in $\mathcal{S}_0$ (that is, $rog(g) \leq 0$). By induction on r, we have that $|f(s,a)| = |s|$, and $|f(s,ra)| = |g(f(s,r))| \leq |f(s,r)| \leq \max(|s|,|r|)$.

Step. Given $f \in \mathcal{S}_{p+1}$, defined by SREC in g and h in $\mathcal{S}_p$, we have

$$\begin{aligned} |f(s,t,a)| &= |g(s,t)| && \text{by definition of } f \\ &\leq |\max(|s|,|t|)| && \text{by inductive hypothesis.} \end{aligned}$$

and

$$\begin{aligned} |f(s,t,ra)| &= |h(f(s,t,r),t,ra)| \\ &\leq |\max(|f(s,t,r)|,|t|,|ra|)| \\ &\leq |\max(\max(|s|,|t|,|r|),|t|,|ra|)| \\ &\leq |\max(|s|,|t|,|ra|)|. \end{aligned}$$

by definition of f, inductive hypothesis on h and induction on r. ■

Lemma 5.2: Each f in $\mathcal{TS}_{qp}$ (with $p,q \geq 1$) can be computed by a register machine within time $O(n^p)$ and space $O(n^q)$.

Proof: Let f be in $\mathcal{TS}_{qp}$. By definition 5.2, f is defined by weak composition of $h \in \mathcal{T}_q$ into $g \in \mathcal{S}_p$, that is, $f(s,t,r) = g(h(s,t,r),t,r)$. The theorem 4.1 states that there exists a register machine M_h which computes h within time n^q, and there exists another register machine M_g which computes g within time n^p. Since g belongs to $\mathcal{S}_p$, lemma 5.1 holds for g; hence, the space needed by M_g is at most n.
Define now a machine M_f that, by input s,t,r, performs the following steps:
(1) it calls M_h on input s,t,r;
(2) it calls M_g on input $h(s,t,r),t,r$, stored in the appropriate registers.
According to lemma 4.1, M_h needs time equal to $|s| + lh(h)(|t|+|r|)^q$ to compute h, and M_g needs $|h(s,t,r)| + lh(g)(|t|+|r|)^p$ to compute g.

This happens because lemma 4.1 shows, in general, that the time used by a register machine to compute a program is bounded by a polynomial in the length of its inputs, but, more precisely, it shows that the time complexity is linear in $|s|$. Moreover, since in our language there is no kind of identification of x as z, M_f never moves the content of a register associated to $h(s,t,r)$ into another register and, in particular, into a register whose value plays the role of recursive counter. Thus, the overall time-bound is $|s|+lh(h)(|t|+|r|)^q+lh(g)(|t|+|r|)^p$ which can be reduced to n^p, being $q \leq p$.
M_h requires space n^q to compute the value of h on input s,t,r; as we noted above, the space needed by M_g for the computation of g is linear in the length of the input, and thus the overall space needed by M_f is still n^q. ■

Lemma 5.3: A register machine which computes its output within time $O(n^p)$ and space $O(n^q)$ can be simulated by an $f \in \mathcal{TS}_{qp}$.

Proof: Let M be a register machine, whose computation is time-bounded by n^p and, simultaneosly, it is space-bounded by n^q. M can be simulated by the composition of two machines, M_h (time-bounded by n^q), and M_g (time-bounded by n^p and, simultaneously, space-bounded by n): the former delimits (within n^q steps) the space that the latter will successively use in order to simulate M.
By theorem 4.1 there exists $h \in \mathcal{T}_q$ which simulates the behaviour of M_h, and there exists $g \in \mathcal{T}_p$ which simulates the behaviour of M_g; this is done by means of nxt_g, which belongs to $\mathcal{S}_0$, since it never adds a digit to the description of M_g without erasing another one.
According to the proof of lemma 4.2, we are able to define $\sigma_{n-1} \in \mathcal{S}_n$, such that $\sigma_{n-1}(s,t) = nxt_g^{|t|^n}$. The result follows defining $sim(s) = \sigma_{p-1}(h(s),s) \in \mathcal{TS}_{qp}$. ■

Theorem 5.1: f belongs to $\mathcal{TS}_{qp}$ if and only if f is computable by a register machine within time $O(n^p)$ and space $O(n^q)$.

Proof: By lemma 5.2 and lemma 5.3. ■

VI. Conclusions

We have provided a resource-free characterization of register machines that computes their output within polynomial time, and we have extended it to register machines that computes their output with a polynomial bound imposed on time and space, simultaneously; this is made by a version of predicative recursion and a related recursive programming language. A program written in this languages can be analyzed, and the complexity of the program is evaluated by means of this simple analysis. This result represents a preliminary step for a resource-free classification of the hierarchy of time-space classes between PTIME and PSPACE, as defined in [3].

REFERENCES

[1] S. Bellantoni and S. Cook, "A new recursion-theoretic characterization of the poly-time functions," Computational Complexity, vol. 2, 1992, pp. 97-110.

[2] S. Caporaso, G. Pani, E. Covino, "A Predicative Approach to the Classification Problem," Journal of Functional Programming, vol. 11(1), Jan. 2001, pp. 95-116.

[3] P. Clote, "A time-space hierarchy between polynomial time and polynomial space," Math. Sys. The., vol. 25, 1992, pp. 77-92.

[4] A. Cobham, "The intrinsic computational difficulty of functions," in Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science. North-Holland, Amsterdam, 1962, pp. 24-30, Y. Bar-Hillel Ed.

[5] M. Hofmann, "Linear types and non-size-increasing polynomial time computation," Information and Computation, vol. 183(1), May 2003, pp. 57-85.

[6] D. Leivant, "A foundational delineation of computational feasibility," in Proceedings of the Sixth Annual IEEE symposium on Logic in Computer Science (LICS'91). IEEE Computer Society Press, 1991, pp. 2-11.

[7] D. Leivant, "Stratified functional programs and computational complexity," in Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93). ACM, New York, 1993, pp. 325-333.

[8] D. Leivant, Ramified recurrence and computational complexity I: word recurrence and polytime. Birkauser, 1994, pp. 320-343, in Feasible Mathematics II, P. Clote and J. Remmel Eds.

[9] D. Leivant and J.-Y. Marion, "Ramified recurrence and computational complexity II: substitution and polyspace," Computer Science Logic, LNCS vol. 933, 1995, pp. 486-500, J. Tiuryn and L. Pocholsky Eds.

[10] I. Oitavem, "New recursive characterization of the elementary functions and the functions computable in polynomial space," Revista Matematica de la Univaersidad Complutense de Madrid, vol 10.1, 1997, pp. 109-125.

[11] H. Simmons, "The realm of primitive recursion," Arch.Math. Logic, vol. 27, 1988, pp. 177-188.

Hardware Realization of Robust Controller Designed Using Reflection Vectors

Ján Cigánek, Michal Kocúr, Štefan Kozák
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava
Bratislava, Slovakia
E-mail: jan.ciganek@stuba.sk, michal.kocur@stuba.sk, stefan.kozak@stuba.sk

***Abstract*— The presented paper deals with the new approach of robust controller design using the reflection vectors techniques. The control structure consists of feed-forward and feedback part. Proposed algorithms were tested using Field-Programmable Gate Array (FPGA) technology for DC motor. Simulations and co-summations were realized in MATLAB-Simulink. The co-simulation allows as to validate working hardware and to accelerate simulations in Simulink and MATLAB. The obtained results demonstrate very effective applicability of the theoretical principles for control of processes subject to parametrical model uncertainty.**

Keywords - robust control; robust stability; parametrical uncertainty; quadratic programming; reflection vectors; FPGA; co-simulation.

I. Introduction

During the last ten years, development of robust control elementary principles and evolution of new robust control methods for different model uncertainty types are visible. Based on theoretical assumptions, modeling and simulation methods, an effective approach to the control of processes with strong and undefined uncertainties is designed. Such uncertainties are typical for biotechnology processes, chemical plants, automobile industry, aviation, etc. For such processes, it is necessary to design robust and practical algorithms which ensure the high performance and robust stability using proposed mathematical techniques with respect the parametric and unmodelled uncertainties [1][2]. Solution to such problems is possible using robust predictive methods and „soft-techniques“ which include fuzzy sets [3], neuron networks and genetic algorithms.

Robust control is used to guarantee stability of plants with parameter changes. The robust controller design consists of two steps:

- analysis of parameter changes and their influence for closed-loop stability,
- robust control synthesis.

There are two approaches for implementing control systems using digital technology. The first approach is based on software which implies a memory-processor interaction. The memory holds the application program while the processor fetches, decodes, and executes the program instructions. Programmable Logic Controllers (PLCs), microcontrollers, microprocessors, Digital Signal Processors (DSPs) and general purpose computers are tools for software implementation. On the other hand, the second approach is based on hardware. Early hardware implementation is achieved by magnetic relays extensively used in old industry automation systems. Then, it became achievable by means of digital logic gates and Medium Scale Integration (MSI) components. When the system size and complexity increases, Application Specific Integrated Circuits (ASICs) are utilized. The ASIC must be fabricated on a manufacturing line, a process that takes several months, before it can be used or even tested [4][5]. FPGAs are configurable ICs and used to implement logic functions.

Today’s high-end FPGAs can hold several millions gates and have some significant advantages over ASICs. They ensure ease of design, lower development costs, more product revenue and the opportunity to speed products to market [6]. At the same time, they are superior to software-based controllers as they are more compact, power-efficient, while adding high speed capabilities.

The presented paper is organized as follows. In Section II, the complete procedure of robust controller design using reflection vectors is presented. Section III offers a short overview of hardware implementation of the proposed control algorithm using FPGA. The applicability of the control algorithm is shown on the case study for a real DC system with parametrical model uncertainty in Section IV. In Section V, the summary of the paper is discussed.

II. Problem Statement

Let us consider the robust control synthesis of a scalar discrete-time control loop. The transfer function of the original continuous-time system is described by the transfer function

$$G_P(s)=\frac{\bar{B}(s)}{\bar{A}(s)}e^{-Ds}=\frac{\bar{b}_m s^m+\bar{b}_{m-1}s^{m-1}+\ldots+\bar{b}_0}{\bar{a}_n s^n+\bar{a}_{n-1}s^{n-1}+\ldots+\bar{a}_0}e^{-Ds} \tag{1}$$

The transfer function of (1) can be converted to its discrete-time counterpart

$$G_P\left(z^{-1}\right)=\frac{b_0+b_1z^{-1}+\ldots+b_nz^{-n}}{1+a_1z^{-1}+\ldots+a_nz^{-n}}z^{-d} \tag{2}$$

For (2), a discrete-time controller is to be designed in form

$$G_R(z) = \frac{q_0 + q_1 z^{-1} + \ldots + q_\upsilon z^{-\upsilon}}{1 + p_1 z^{-1} + \ldots + p_\mu z^{-\mu}} = \frac{Q(z)}{P(z)} = \frac{U(z)}{E(z)} \quad (3)$$

The corresponding closed-loop characteristic equation is

$$1 + G_P(z^{-1})G_R(z^{-1}) = 0 \quad (4)$$

Substituting (3) and (2) in (4), after a simple manipulation yield the characteristic equation

$$\begin{aligned} 1 + G_P G_R &= (1 + p_1 z^{-1} + \ldots + p_\mu z^{-\mu})(1 + a_1 z^{-1} + \ldots + a_n z^{-n}) + \\ &+ (q_0 + q_1 z^{-1} + \ldots + q_\upsilon z^{-\upsilon})(b_1 z^{-1} + \ldots + b_n z^{-n}) z^{-d} = 0 \end{aligned} \quad (5)$$

Unknown coefficients of the discrete controller can be designed using various methods. In this paper, a robust controller design method based on reflection vectors is used.

The pole assignment problem is as follows: find a controller $G_R(z)$ such that $C(z)=e(z)$ where $e(z)$ is a given (target) polynomial of degree k. It is known [7] that, when $\mu=n-1$, the above problem has a solution for arbitrary $e(z)$ whenever the plant has no common pole-zero pairs. In general, for $\mu < n - 1$ exact attainment of a desired target polynomial $e(z)$ is impossible.

Let us relax the requirement of attaining the target polynomial $e(z)$ exactly and enlarge the target region to a polytope V in the polynomial space containing the point e representing the desired closed-loop characteristic polynomial. Without any restriction, we can assume that $a_n = p_0 = 1$ and deal with monic polynomials $C(z)$, i.e., $\alpha_0 = 1$.

Let us introduce the stability measure as $\rho = c^T c$, where

$$c = S^{-1} C \quad (6)$$

and S is a matrix of dimensions $(n + \mu + 1) \times (n + \mu + 1)$ representing vertices of the target polytope V. For monic polynomials holds

$$\sum_{i=1}^{k+1} c_i = 1 \quad (7)$$

where $k = n + \mu$. If all coefficients are positive, i.e., $c_i > 0$, $i = 1,..., k + 1$, then the point C is placed inside the polytope V.

The minimum ρ is attained if

$$c_1 = c_2 = \ldots = c_{k+1} = \frac{1}{k+1} \quad (8)$$

then the point C is placed in centre of the polytope V.

In the matrix form, we have

$$C = Gx \quad (9)$$

where G is the Sylvester matrix of the plant with dimensions $(n + \mu + d + 1) \times (\mu + \upsilon + 2)$ and x is the $(\mu + \upsilon + 2)$-vector of controller parameters: $x = [p_\mu, \ldots, p_1, 1, q_\upsilon, \ldots, q_0]^T$.

Now, we can formulate the following control design problem: find a discrete controller, where the closed-loop characteristic polynomial $C(z)$ is placed:

a) In a stable target polytope V, $C(z) \in V$ (to guarantee stability),
b) As close as possible to a target polynomial $e(z)$, $e(z) \in V$ (to guarantee performance).

Let the polytope V denote the $(k+1) \times N$ matrix composed of coefficient vectors v_j, $j=1,\ldots,N$ corresponding to vertices of the polytope V.

Then, we can formulate the above controller design problem as an optimization task: Find x that minimizes the cost function

$$J_1 = \min_x x^T G^T G x - 2e^T G x = \min_x \|Gx - e\|^2 \quad (10)$$

subject to the linear constraints

$$Gx = V\,w(x), \quad (11)$$

$$w_j(x) > 0,\ j = 1,\ldots, N, \quad (12)$$

$$\sum_j w_j(x) = 1. \quad (13)$$

Here, $w(x)$ is the vector of weights of the polytope V vertices to obtain the point $C = G\,x$. Fulfillment of the latter two constraints (12), (13) guarantees that the point C is indeed located inside the polytope V. Then, finding the robust pole-placement controller coefficients represents an optimization problem that can be solved using the Matlab Toolbox OPTIM (quadprog) with constraints.

Generally J_1 is a kind of distance to the centre of the target polytope V. It is better to use another criterion J_2, which measures the distance to the Schur polynomial $E(z)$

$$J_2 = (C - E)^T (C - E) = (Gx - E)^T (Gx - E). \quad (14)$$

It is possible to use the weighted combination of J_1 and J_2

$$J = (1-\alpha)J_1 + \alpha J_2, \quad 0 \le \alpha \le 1 \quad (15)$$

and to solve the following quadratic programming task

$$\begin{aligned} &J = \min_x \{ x^T G^T [(1-\alpha)(SS^T)^{-1} + \alpha I_{k+1}] Gx - 2\alpha E^T Gx \}, \\ &S^{-1}Gx < 0. \end{aligned} \quad (16)$$

Assume the discrete robust controller design task with parametrical uncertainties in system description. Let us also assume that coefficients of the discrete-time system transfer functions a_n , ..., a_1 and b_n , ..., b_1 are placed in polytope W with the vertices $d^j = \left[a_n^j, \ldots a_1^j, b_n^j, \ldots, b_1^j\right]$:

$$W = conv\{d_j, j = 1, \ldots, M\} \qquad (17)$$

As (9) is linear in system parameters, it is possible to claim that for arbitrary vector of the controller coefficients x is the vector of the characteristic polynomial coefficients $C(z)$ placed in the polytope A with vertices $a^1, \ldots, a^M$:

$$A = conv\{a^j, j = 1, \ldots, M\} \qquad (18)$$

where $a^j = D^j x$ and D^j is the Sylvester matrix of dimensions $(n + \mu + d + 1) \times (\mu + \upsilon + 2)$, composed of vertices set d^j, as in case of the exact model (9).

A. Stable Region Computation via Reflection Coefficients

Polynomials are usually specified by their coefficients or roots. They can be characterized also by their reflection coefficients using Schur-Cohn recursion.

Let $C_k(z^{-1})$ be a monic polynomial of degree k with real coefficients $c_i \in R$, $i = 0, \ldots, k$

$$C(z^{-1}) = 1 + c_1 z^{-1} + \ldots + c_k z^{-k} \qquad (19)$$

Reciprocal polynomial $C_k^*(z^{-1})$ of the polynomial $C_k(z^{-1})$ is defined in [8] as follows

$$C_k^*(z^{-1}) = c_k + c_{k-1} z^{-1} + \ldots + c_1 z^{-k+1} + z^{-k} \qquad (20)$$

Reflection coefficients r_i, $i = 1, \ldots, k$, can be obtained from the polynomial $C_k(z^{-1})$ using backward Levinson recursion [9]

$$z^{-1} C_{i-1}(z^{-1}) = \frac{1}{1 - \left|r_i^2\right|} \left[C_i(z^{-1}) - r_i C_i^*(z^{-1})\right] \qquad (21)$$

where $r_i = -c_i$ and c_i is the last coefficient of $C_i(z^{-1})$ of degree i. From (21) we obtain in a straightforward way:

$$C_i(z^{-1}) = z^{-1} C_{i-1}(z^{-1}) + r_i C_{i-1}^*(z^{-1}). \qquad (22)$$

Expressions for polynomial coefficients $C_{i-1}(z^{-1})$ and $C_i(z^{-1})$ result from equations (22,23):

$$C_{i-1}(z^{-1}) = \frac{1}{1 - \left|r_i^2\right|} \left[\sum_{j=0}^{i-1} \left(c_{i,j+1} - r_i c_{i,i-j-1}\right) z^{-j})\right] \qquad (23)$$

$$C_i(z^{-1}) = \sum_{j=0}^{i} \left(c_{i-1,j-1} + r_i c_{i-1,i-j-1}\right) z^{-j}. \qquad (24)$$

The reflection coefficients r_i are also known as Schur-Szegö parameters [8], partial correlation coefficients [11] or k-parameters [10]. Presented forms and structures were effectively used in many applications of signal processing [10] and system identification [11]. A complete characterization and classification of polynomials using their reflection coefficients instead of roots (zeros) of polynomials is given in [8].

The main advantage of using reflection coefficients is that the transformation from reflection to polynomial coefficients is very simple. Indeed, according to (22) and (24), polynomial coefficients c_i depend multilinearly on the reflection coefficients r_i. If the coefficients $c_i \in R$ are real, then also the reflection coefficients $r_i \in R$ are real.

Transformation from reflection coefficients $r_i, i=1,\ldots,k$, to polynomial coefficients $c_i, i=1,\ldots,k$, is as follows

$$\begin{gathered} c_i = c_i^{(k)}, \\ c_i^{(i)} = -r_i, \\ c_j^{(i)} = c_j^{(i-1)} - r_i c_{i-j}^{(i-1)} \\ i = 1, \ldots, k;\ j = 1, \ldots, i-1 \end{gathered} \qquad (25)$$

<u>Lemma 1.</u> A linear discrete-time dynamic system is stable if its characteristic polynomial is Schur stable, i.e., if all its poles lie inside the unit circle.

The stability criterion in terms of reflection coefficients is as follows [8].

<u>Lemma 2</u>. A polynomial $C(z^{-1})$ has all its roots inside the unit disk if and only if $|r_i| < 1$, $i = 1, \ldots, k$.

A polynomial $C(z^{-1})$ lies on the stability boundary if some $r_i = \pm 1$, $i = 1, \ldots, k$. For monic Schur polynomials, there is a one-to-one correspondence between $C = [c_k, \ldots, c_1]^T$ and $r = [r_1, \ldots, r_k]^T$.

Stability region in the reflection coefficient space is simply the k-dimensional unit hypercube $R = \{r_i \in (-1,1), i = 1, \ldots, k\}$. The stability region in the polynomial coefficient space can be found starting from the hypercube R.

B. Stable Polytope of Reflection Vectors

It will be shown that, for a family of polynomials, the linear cover of the so-called reflection vectors is Schur stable.

<u>Definition 1.</u> The reflection vectors of a Schur stable monic polynomial $C(z^{-1})$ are defined as the points on stability boundary in polynomial coefficient space generated by changing a single reflection coefficient r_i of the polynomial $C(z^{-1})$.

Let us denote the positive reflection vectors of $C(z^{-1})$ as $v_i^+(C) = (C|r_i = 1), i = 1,\dots,k,$ and the negative reflection vectors of $C(z^{-1})$ as $v_i^-(C) = (C|r_i = -1), i = 1,\dots,k.$

The following assertions hold:

1) every Schur polynomial has $2k$ reflection vectors $v_i^+(C)$ and $v_i^-(C), i = 1,\dots,k;$
2) all reflection vectors lie on the stability boundary $(r_i^v = \pm 1);$
3) the line segments between reflection vectors $v_i^+(C)$ and $v_i^-(C)$ are Schur stable.

In the following theorem a family of stable polynomials is defined such that the polytope generated by reflection vectors of these polynomials is stable.

<u>*Theorem 1*</u>**.** Consider $r_1^C \in (-1,1),$ $r_k^C \in (-1,1)$ and $r_2^C = \dots = r_{k-1}^C = 0$. Then, the inner points of the polytope *V(C)* generated by the reflection vectors of the point *C*

$$V(C) = conv\{v_i^{\pm}(C), i = 1,\dots,k\} \qquad (26)$$

are Schur stable.

C. Robust Controller Design

A robust controller is to be designed such that the closed-loop characteristic polynomial is placed in the stable polytope (linear cover) of reflection vectors. It means that the following problems have to be solved:

1. choice of initial polynomial $C(z^{-1})$ for generating the polytope *V(C),*
2. choice of $k + 1$ most suitable vertices of *V(C)* to build a target simplex *S,*
3. choice of a target polynomial $E(z^{-1})$.

In the following section some "thumb rules" are given for choosing a stable target simplex *S*.

To choose $k + 1$ vertices of the target simplex *S* we use the well known fact that poles with positive real parts are preferred to those with negative ones [1]. The positive reflection vectors $v_i^+(C)$ with *i* odd and negative reflection vectors $v_i^-(C)$ with *i* even are chosen yielding *k* vertices. The *(k+1)*th vertex of the target simplex *S* is chosen as the mean of the remaining reflection vectors.

The target polynomial $E(z^{-1})$ of order k is reasonable to be chosen inside the stable polytope of reflection vectors *V(C).* A common choice is $E(z^{-1})=C(z^{-1})$.

For higher-order polynomials, the size of the target simplex *S* is considerably less than the volume of the polytope of reflection vectors *V*. That is why the above quadratic programming method with a preselected target simplex *S* works only if uncertainties are sufficiently small. Otherwise, it is reasonable to use some search procedure to find a robust controller such that the polytope of closed-loop characteristic polynomial is placed inside the stable polytope of reflection vectors *V(C).*

In terms of the performance, the comparison of the proposed solution with other solutions would not be quite transparent due to different structure of the control loops or due to the different polynomial degrees of the controller.

III. Implementation of Control Algorithm

The digital form of the controller can be obtained from (3). Recursive form of control algorithm is expressed by the following equation:

$$u(k) = q_1e(k-1) + q_2e(k-2) - p_1u(k-1) - p_2u(k-2) \qquad (27)$$

For implementation of control algorithm (27) for FPGA it is necessary to decompose equation into simple arithmetic operations:

$$\begin{aligned} e(k) &= w(k) - y(k) \\ q_1e_{-1} &= q_1 * e(k-1) \\ q_2e_{-2} &= q_2 * e(k-2) \\ p_1u_{-1} &= p_1 * u(k-1) \\ p_2u_{-2} &= p_2 * u(k-2) \\ s_1 &= q_1e_{-1} + q_2e_{-2} \\ s_2 &= p_1u_{-1} + p_2u_{-2} \\ s_3 &= s_1 - s_2 \end{aligned} \qquad (28)$$

Control output *u* must be bounded in the range from u_{min} to u_{max}.

$$u(k) = \begin{matrix} u_{\max} \rightarrow if(s3 > u_{\max}) \\ s3 \rightarrow if(u_{\min} \le s3 \le u_{\max}) \\ u_{\min} \rightarrow if(s3 < u_{\min}) \end{matrix} \qquad (29)$$

In this case, the parallel design of control algorithm is used, which means that each of the operation has its own arithmetic unit, either accumulator or multiplier. Parallel design is shown in Figure 1.

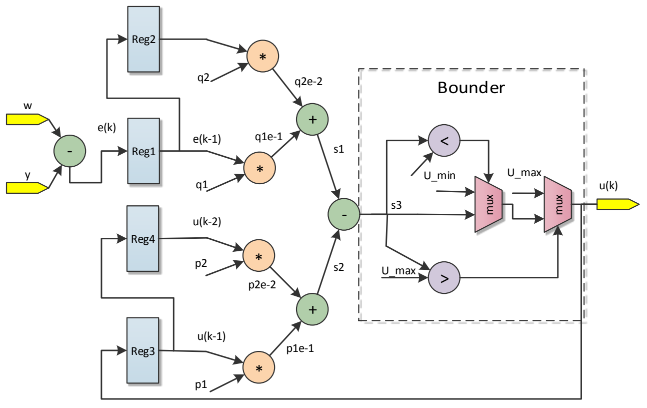


Figure 1. Parallel design of control algorithm

Each sampling period is loaded the motor system output *y(k)* from the input in. Control error *e(k)* is computed in sub block where the signal *y(k)* is subtracted from *w(k)*. Signal *e(k)* is held in the registry *REG1* for one sampling period. Register *REG1* output signal is thus *e(k-1)*. In the same manner, *e(k−2), u(k−1)* and *u(k−2)* are recorded at *REG2*, *REG3* and *REG4* by latching *e(k−1), u(k)* and *u(k-1)* respectively. For multiplication, they are using digital signal processor (DSP) cores in FPGA chip. Results of multiplications are counted in to control output. This control output is then bounded in the range from −12V to 12V.

Inputs *w* and *y* are represented with rpm (Revolutions per minute). Input range is -2048 to 2047 rpm, because of the 12bit signed data type. Output of the controller is represented with volts. In signed binary representations the maximum control output is $12_{(10)}$V = $01100_{(2)}$ and the minimum is $-12_{(10)}$ V = $10100_{(2)}$. We can write this range into 5 bits. Real numbers are useful for better quantization of the control output. For implementation of the real numbers, it has been used fixed point arithmetic [12]. As we can see in the Figure 2, the first (MSB) bit of output vectors is reserved for a sign. Next four bits are reserved for the integer part and last seven bits are used for the fractional part.

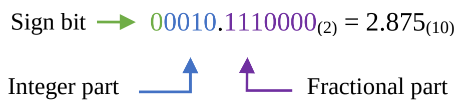


Figure 2. Fixed-point control output

Fixed point arithmetic is applied throughout the control algorithm. In designing this algorithm, the fixed-point arithmetic range rules must be respected. The data widths in the fixed-point arithmetic were designed that there is no possibility of an overflow. For example, the result of summation or subtraction of two 12-bit vectors has range 13-bit. Table 1 represents used range rules for fixed point arithmetic.

TABLE I. FIXED-POINT RANGE RULES

Operation	Result Range
A + B	Max(A'left, B'left)+1 Min(A'right, B'right)
A – B	Max(A'left, B'left)+1 Min(A'right, B'right)
A * B	A'left + B'left+1 A'right + B'right

In the case of parallel the design of control algorithm, the control output after last summation (resp. subtraction) has range 40-bit (16-bit for fractional part). It must be used a bounder block to ensure of range (-12 V to 12 V) for 12bit. Bounder is the value limitation logic that keeps the output in the defined range. Bounded signal is latched at register *REG3*, thus becomes *u(k−1)* of next cycle. In this way, the anti-windup protection is also ensured.

System Generator toolbox for Simulink ensures that between the blocks gateway in and gateway out algorithm performs as it was implemented on FPGA (Figure 3). The parallel design of control algorithm designed in VHDL is contained in the control algorithm block.

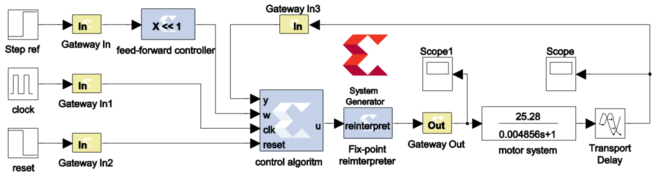


Figure 3. Schematic of control circuit using Xilinx blocks

In the co-simulation process, we used Xilinx Spartan-6 FPGA which is included in SP-601 demoboard. Spartan-6 FPGAs offer advanced power management technology, up to 150K logic cells, advanced memory support, 250MHz DSP slices, and 3.2Gbps low-power transceivers [13].

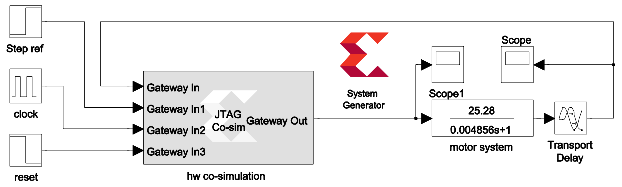


Figure 4. Schematic of control circuit using Co-sim block

Based on the successful verification of the Xilinx blocks algorithm, we generated the co-simulation block by Xilinx System Generator (Figure 4). If is co-sim block included in simulation scheme it must be FPGA connected during the simulation process into the computer. At the start of simulation System Generator records functionality of co-sim block into FPGA. The co-sim block behaves as an in-out black box. Control output is computed in FPGA. Other blocks like the transfer function or step generator are still simulated in Simulink. Communication between FPGA board and the computer can be provided by Point to Point Ethernet or USB JTAG.

IV. CASE STUDY

Let us consider a DC system described by the first order nominal transfer function

$$G_P(s)=\frac{B(s)}{A(s)}e^{-Ds}=\frac{K}{T_1 s+1}e^{-Ds}=\frac{25.28}{0.004856s+1}e^{-0.001s} \qquad (30)$$

where the coefficients *K*, T_1 are varying in uncertainty intervals $K\in\langle 25;25.5\rangle$, $T_1\in\langle 0.0045;0.0052\rangle$

Let us consider the nominal continuous-time transfer function which is converted to the discrete-time form with the sampling period *T=0.001s:*

$$G_p(z^{-1})=\frac{0.4228z^{-1}+4.282z^{-2}}{1-0.8139z^{-1}} \qquad (31)$$

The main task is to design a robust iscrete-time controller (3), with polynomial degrees $\upsilon=1, \mu=2$.

From the transfer function (31) and matrix form of (9), we can obtain:

$$C=\begin{bmatrix} 0 & 0 & 0 & 0.4228 & 0 \\ -0.8139 & 0 & 0 & 0 & 0.4228 \\ 1 & -0.8139 & 0 & 0 & 0 \\ 0 & 1 & -0.8139 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} p_2 \\ p_1 \\ 1 \\ q_1 \\ q_0 \end{bmatrix}$$

Let us choose the initial polynomial *C(z^{-1})* for generating the polytope *V(C)* as follows

$$C(z)=[1-0.2z^{-1}][1-0.1z^{-1}][1+0.2z^{-1}][1+0.3z^{-1}] \qquad (32)$$

with reflection coefficients $r_1=0.2,\ r_2=-0.07,\ r_3=-0.008,$ $r_4=0.0012.$

Now, we can find the reflection vectors $v_i\,(C)$ of the initial polynomial *C(z^{-1})* leading to the matrix form of the target simplex *S* (vertex polynomial coefficients)

$$S=\begin{bmatrix} 0 & 0 & 0 & -0.3 & 0.5 \\ 0 & 0 & -0.3 & 0.5 & 0 \\ 0 & -0.3 & 0.5 & 0 & 0 \\ -0.3 & 0.5 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad (33)$$

The discrete-time controller design task for the nominal transfer function (30) has been solved via quadratic programming taking *α=0.3* in the cost function *J* (16).

For the selected target simplex *S*, we have obtained the following discrete-time feedback controller

$$G_{FB}\left(z^{-1}\right)=\frac{Q(z^{-1})}{P(z^{-1})}=\frac{0.025+0.0231z^{-1}}{1+0.0159z^{-1}+0.013z^{-2}} \qquad (34)$$

and the control law is expressed in recursive form

$$\begin{aligned} u_2(k)=-0.016u_2(k-1)-0.013u_2(k-2)+ \\ +0.025y(k)+0.0231y(k-1) \end{aligned} \qquad (35)$$

For verification of the FGPA hardware co-simulation of the digital controller, we realized a practical experiment. In co-simulation, we made step of reference signal at 0.01s.

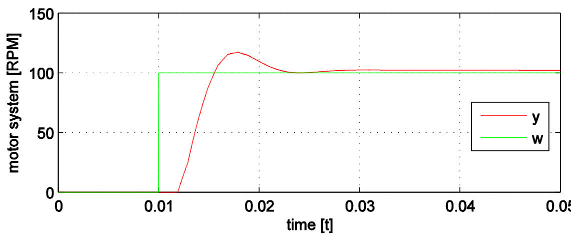


Figure 5. Time response of output and reference variable under robust controller

The corresponding closed-loop step response under the feedback controller (34) and feed-forward controller $G_{FF}\left(z^{-1}\right)=S(z^{-1})/P(z^{-1})=2$ is in Figure 5.

V. Conclusion

The paper deals with the development of robust control algorithm based on reflection vectors methodology. The proposed algorithm can be effective realized using FPGA structure end guaranteeing stability, robustness and high performance. Theoretical results were verified on the example for feedback and feedforward control structures. The methods were also successfully tested for stable and unstable processes.

The illustrative example was solved using quadratic programming for suitably defined performance function. Simulation results prove applicability of the proposed robust controller design theory for systems with parametric uncertainty.

Digital controller was successfully implemented and hardware co-simulated on Spartan-6 FPGA board. From the obtained results, it is evident that the application of FPGA structure is very suitable for high speed processes. In this paper, we presented the basic necessary principles how to realize and modified existing digital robust control algorithms. The co-simulation option can be useful to accelerate simulation of advanced control algorithms.

Acknowledgment

This paper is supported by APVV project No. APVV-0772-12.

References

[1] J. Ackermann, “Robust Control” Systems with Uncertain Physical Parameters. Springer-Verlag. London, UK, 1993.

[2] J. Ciganek and S. Kozak, „Robust Controller Design Based on Reflection Vectors”. Control Applications, (CCA) & Intelligent Control, (ISIC), 2009, pp. 938–943.

[3] J. Ciganek and F. Noge,” Fuzzy Logic Control of Mechatronic Systems”, IEEE Int. conf. on Process Control, Strbske Pleso, Slovak Republic, 2013, pp. 309–313.

[4] Š. Kozak “Development of control engineering methods and their applications in industry” In 5th Int. Scientific-Technical Conference Process Control 2002. Kouty nad Desnou, Czech Rep., 2002, pp. 218-222.

[5] M. Kocur, “HW Realization of PID algorithm based on FPGA,” Slovak University of Technology in Bratislava, Bratislava.

[6] V. Viswanathan “Embedded Control Using FPGA”. Indian Institute of Technology, Bombay, 2005

[7] L. H. Keel and S. P. Bhattachayya, “A linear programming approach to controller design.” Automatica, vol. 35, 1999, pp. 1717–1724.

[8] J. L. Diaz-Barrero and J. J. Egozcue, “Characterization of polynomials using reflection coefficients.” Appl. Math. E-Notes, vol. 4, 2004, pp. 114–121.

[9] B. Picinbono and M. Bendir, “Some properties of lattice autoregressive filters”, IEEE Trans. Acoust. Speech Signal Process, 34, 1986, pp. 342–349.

[10] A. M. Oppenheim and R. W. Schaffer, “Discrete-Time Signal Processing.” Prentice-Hall, Englewood Cliffs,1989.

[11] S. M. Kay, “Modern Spectral Estimation,” Prentice Hall, New Jersey, 1988.

[12] D. Bishop. “Fixed point package user’s guide”. http://www.vhdl.org, [retrieved: 11, 2014]

[13] Xilinx, “SP601 Hardware User Guide”

Combining Code Refactoring and Auto-Tuning to Improve Performance Portability of High-Performance Computing Applications

Chunyan Wang*†, Shoichi Hirasawa*†, Hiroyuki Takizawa*†, and Hiroaki Kobayashi‡
* Graduate School of Information Sciences, Tohoku University
Sendai, Japan
Email: {wchunyan, hirasawa}@sc.isc.tohoku.ac.jp, takizawa@cc.tohoku.ac.jp
†CREST, Japan Science and Technology Agency
‡Cyberscience Center, Tohoku University
Email: koba@cc.tohoku.ac.jp

***Abstract*—An existing High-Performance Computing (HPC) application is usually optimized for a particular platform to achieve high performance. Hence, such an application is often unable to run efficiently on other platforms, i. e., its performance is not portable. The purpose of this work is to establish a systematic way to improve performance portability of HPC applications, to which various kinds of platform-specific optimizations have already been applied. To this end, we combine code refactoring and auto-tuning technologies, and develop a programming tool for HPC refactoring. Auto-tuning is a promising technology to enable an HPC application to adapt to different platforms. In general, however, an auto-tuning tool is not applicable to an HPC application if the application has already been optimized for a particular platform. In this work, a code refactoring tool that interactively asks a user for necessary information to undo some platform-specific optimizations in an existing application is developed based on Eclipse, and hence auto-tuning techniques can be applied to the application. The evaluation results demonstrate that combining code refactoring and auto-tuning is a promising way to replace platform-specific optimizations with auto-tuning annotations, and thereby to improve the performance portability of an HPC application.**

***Keywords*–auto-tuning; code refactoring; high-performance computing; performance portability.**

I. Introduction

Legacy applications are successful and therefore mature, and likely have been in existence for a long period of time. Thus, legacy applications are often required to migrate to new platforms, to use new algorithms, and to be components in larger systems. Typical platforms are determined by a hardware architecture, an operating system, and runtime libraries. Moore's law [1] has predicted the dramatic improvement of computing hardware. As legacy applications are ported to meet ever-changing requirements, even well-designed applications are subject to structural erosion. The quality of any code base with a long lifespan tends to degrade over time [2]. Maintenance of a legacy application can become an error-prone, time- and resource-consuming work.

In the specific field of HPC, the main concern is to fully utilize the potential of a specific target platform to achieve high performance [3]. Today, an existing HPC application is usually optimized for a particular platform. While HPC systems (hardware and software) reach unprecedented levels of complexity, such an application is often unable to run efficiently on other platforms because different platforms require different optimizations, and hence its performance is not portable. In order to maximize performance, the developer must carefully consider optimizations relevant to each target platform. As a result, tuning for an individual platform is a highly-specialized and time-intensive process. The increasing complexity of HPC systems, their long lifespans, and the plethora of desirable source code optimizations make HPC applications hard to maintain.

Automatic performance tuning, also known as auto-tuning, provides performance portability, as the auto-tuning process can easily be re-run on new platforms which require different sets of optimizations [4]. Auto-tuning is increasingly being used in the domain of HPC to optimize programs. However, auto-tuning is usually designed for programs not optimized for any specific platform. It is difficult to auto-tune an HPC application, to which platform-specific optimizations are already applied.

Refactoring is a promising solution to gradual software decay [2][5]. An application code can be refactored so that it is easier to apply auto-tuning technologies. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [5]. Refactoring tools that can automate the refactoring process have shown their advantages in improving readability, maintainability and extensibility of object-oriented programs. Behavior-preserving code transformations, which are simply called *refactorings* in the research field of code refactoring, can help to improve the code structure, thereby potentially setting the stage for improvements. Despite the potential, refactorings that are specific to HPC applications are rarely provided.

However, some refactorings for HPC applications cannot be fully automated because the information required for those refactorings has been lost when the platform-specific optimizations were applied to the application code. Users have to specify where and how the application code was optimized. Usually, a significant part of the knowledge required to perform the refactoring remains implicitly in the user's head. It is promising to use user knowledge to compensate the lost information that is necessary for the refactoring. Semi-automated refactoring can be a promising approach to refactoring application codes to be tunable with user knowledge.

Photran [6] provides some refactorings, called performance refactorings, to facilitate some loop optimizations (*interchange loop*, *fuse loop*, *reverse loop*, *tile loop* and *unroll loop*) specific to Fortran, which is the backbone of the HPC community [7]. On one hand, these refactorings help to improve the performance of an application. On the other hand, loop refactorings such as loop tiling and loop unrolling make the code difficult to understand, and consequently make it hard for further optimization. Due to the differences in system configurations, an appropriate loop optimization on an HPC application usually changes depending on its target platform [8]. When the application code is ported to a newly available platform, the performance is usually not portable to the new platform. To make matters worse, it is difficult to optimize the refactored code for a new platform. Therefore, those performance refactorings always lead to low performance portability.

The purpose of this work is to establish a systematic way to improve performance portability of HPC applications, to which various kinds of platform-specific optimizations have already been applied. To achieve this goal, we combine code refactoring and auto-tuning technologies. A code refactoring tool is designed to support the process of undoing platform-specific optimizations of an existing HPC application. Note that some information such as the original loop length may be lost by platform-specific loop optimizations such as Photran's performance refactorings. Hence, we develop a refactoring tool that is assumed to be a part of an integrated development environment (IDE) so that the tool can interactively ask the user to specify the missing information for "reverse transformations" of performance refactorings. In this work, the reverse transformations are called *HPC refactorings*. As a result of HPC refactorings, platform-specific optimizations are replaced with annotations for auto-tuning to make the application adaptable to different platforms. Moreover, the resulting application code becomes easy to read and maintain. The evaluation results indicate that the HPC refactoring tool is helpful to support replacement of platform-specific optimizations with auto-tuning annotations, and thereby to improve the performance portability of an HPC application.

The remainder of this paper is organized as follows. Section II introduces the related work. Section III describes the proposed method and illustrates it with two examples. Section IV shows the evaluation of the proposed method. Finally, Section V gives the conclusion of this work, and states future work.

II. Related Work

This section reviews the related researches, and classifies them into four categories: (1) automation of refactorings; (2) code refactorings for HPC applications; (3) platform-specific optimizations that degrade performance portability; and (4) refactoring and performance tuning.

A. Fully-automated versus Semi-automated Refactoring Tools

Mens and Tourwé [9] identified three activities associated with the process of refactoring:

1) Identification of where an application code should be refactored.
2) Determination of which refactoring(s) should be applied.
3) Application of the selected refactorings.

The degree of the automation of a refactoring tool depends on which of the refactoring activities are supported by the tool.

Contemporary IDEs such as Eclipse [10] often support a semi-automated approach to refactoring. Tokuda and Batory's research [11] indicated that a semi-automated approach can drastically increase the productivity in comparison with manual refactoring.

Some researchers demonstrated the feasibility of fully-automated refactoring [12][13]. Guru is a fully automated tool for refactoring inheritance hierarchies and refactoring methods in SELF programs [13]. Optimization techniques that are performed by compilers can also be considered as fully automated refactoring techniques. Although fully automated tools provide the ability to quickly and safely improve the structure of the code without altering its functionality, the lack of user input leads to the introduction of meaningless identifiers and could make the current application become more difficult to understand than before.

Semi-automated refactoring tools involve human interaction to address the problems caused by fully-automated refactoring. The semi-automated refactoring tool may become time-consuming when the application is in large scale. Despite this problem, semi-automated refactoring remains the most useful approach in practice, since a significant part of the knowledge required to perform the refactoring cannot be extracted from the software, but remains implicit in the developer's head [9].

B. Refactorings for HPC Applications

What HPC programmers concern most is to maximize the performance on their target platforms. While refactorings are typically used to improve the readability and maintainability of a code, refactorings specific to Fortran and HPC to improve performance are rarely provided. HPC programs are more difficult to restructure because data flow and control flow are tightly interwoven [9]. Because of this, restructurings are typically limited to the level of a function or a block of code [14].

Nevertheless, many refactorings that improve performance still have to be done manually. The parser and general language infrastructure of a refactoring tool and performance preserving requirements make it still a great challenge to develop a new refactoring tool for the specific domain of HPC. Great efforts have been made to develop tools to restructure the Fortran and HPC codes [3][15].

Bodin et al. built an object-oriented toolkit and class library for building Fortran and C++ restructuring tools [16]. It requires complete understanding of the internal parser structures for users to add language extensions to Fortran or C. CamFort [17] is a tool that provides automatic refactoring for improving the code quality of existing models. SPAG [18] is a restructuring tool, which unscrambles spaghetti Fortran66 code, and convert it to structured Fortran 77. New projects such as Eclipse Parallel Tools Platform (PTP) have recently made strides to address the needs of HPC developers [15]. It provides some refactorings such as Rename and Extract Method. As a component of the Eclipse PTP, Photran provides an IDE and refactoring tool for Fortran. However, since Photran was originally designed to improve maintainability or performance [19], it does not go far enough towards solving the performance portability problem.

C. Platform-Specific Optimizations

Some researchers have indicated that platform-specific optimizations degrade the performance portability of application codes. Ratzinger et al. [20] exploited historical data extracted from repositories, and pointed out that certain design fragments in software architectures can have a negative impact on system maintainability. Then, they proposed an approach to detecting such design problems to improve the evolvability of an application. Bailey et al. [21] pointed out that, an application may not be able to run efficiently with available computer resource unless the application has been optimized for the particular system. Our previous research [22] has also reported that platform-specific optimizations have a strong negative impact on performance portability of an existing HPC application.

D. Refactoring and Performance Tuning

Fowler [5] stated that refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning. Du et al. [23] claimed that if the architectural details are known, auto-tuning is an effective way to generate tuned kernels that deliver acceptable levels of performance. Moore's research [24] applied *Extract Procedure* refactoring to the performance critical regions to facilitate performance tuning. The research also shows the possibility to enhance refactoring tools with auto-tuning techniques. In [25], it is verified that OpenACC directives for accelerators provide a mechanism to stay in a current high-level language. OpenACC directives are expected to enable programmers to easily develop portable applications that maximize the performance with the hybrid CPU/GPU architecture, and accelerate existing applications quickly by adding a few lines of code. However, platform-specific optimizations are still required even if OpenACC directives are adopted for accelerator computing [26].

III. The Proposed HPC Refactoring Tool

In this work, we develop an HPC refactoring tool to improve performance portability of HPC applications. Section III describes the methodology of the proposed HPC refactoring tool. This method provides an undo mechanism with code refactoring to undo platform-specific optimizations to make an application code tunable, and a redo mechanism of the optimizations with an auto-tuning technique. Undoing of optimizations is necessary because auto-tuning tools usually assume un-optimized codes as their inputs. By using auto-tuning to make an existing HPC application adaptive to other platforms, the performance portability of the application can be improved in a systematic way.

A. Overview of the Proposed Method

Given an application that is optimized for a specific platform for high performance, the performance portability of this application is the ability to retain the performance when the application is ported to other platforms.

Figure 1 illustrates the proposed method to improve performance portability. *Original Program* represents the source program that has already been optimized for a specific platform. In this work, we develop a code refactoring tool as a plug-in of Eclipse IDE to automate the process of undoing platform-specific optimizations such as loop unrolling with a certain unroll factor. With our refactoring tool, *Original Program* is refactored to *Refactored Program*, in which platform-specific optimizations are undone. Hence, an auto-tuning technique can be applied to redo the optimizations to achieve high performance on various platforms.

While it is difficult to auto-tune *Original Program*, *Refactored Program* can use an auto-tuning technique because such a technique is usually designed for programs not optimized for a specific platform. As a result, *Autotuned Program* that is *Refactored Program* with an auto-tuning technique can

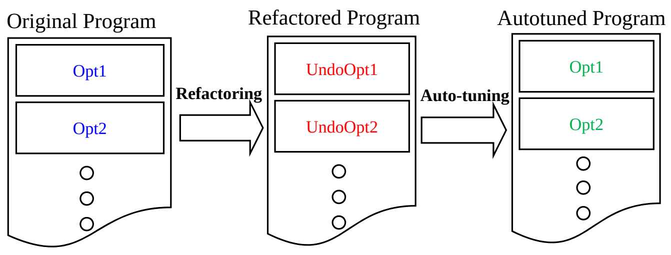


Figure 1. Overview of the proposed method for improving performance portability.

achieve high performance portability across multiple platforms while maintaining the performance on the original platform, for which *Original Program* was optimized. In this way, we systematically improve the performance portability of an existing HPC application.

The code refactoring tool can undo a platform-specific optimization only if the transformation rule of the optimization is already known. Performance refactorings, such as loop tiling and loop unrolling, provided by Photran degrade the performance portability of an application code. Since the transformation rule of these refactorings are already known, we will use these refactorings to explain the undo mechanism and show how the proposed method can contribute to the performance portability.

B. Undo Mechanism of the Proposed Method

The undo mechanism provides a semi-automated refactoring tool to allow users to specify the necessary information for refactoring. Users select the code region and determine which refactoring should be applied. The refactoring tool then checks the preconditions for behavior preserving. Once the preconditions are guaranteed, users are asked to input necessary information for applying corresponding refactoring.

1) Undo Loop Unrolling: Loop unrolling is a loop transformation technique that attempts to optimize a program's performance by reducing instructions that control the loop [27].

The general form of the do-loop is as follows.

```
do var = expr1, expr2, expr3
   statements
end do
```

Here, *var* is the loop index, *expr1* specifies the initial value of *var*, *expr1* is the upper bound, and *expr3* is the increment (step). The variable defined in the do-statements is incremented by 1 by default.

Loop unrolling replicates the code inside a loop body multiple times. The *step* that determines the number of replications is called an *unroll factor*.

The *unroll loop* refactoring provided by Photran applies loop unrolling to the outermost loop of the selected nested do-loop. Figure 2 shows a typical do-loop unrolled by *unroll loop* refactoring when the selected do-loop is the innermost loop and the unroll factor is set to be "*B*."

To perform an undoing operation, users have to specify the do-loop nest that loop unrolling was applied to and the unroll factor prior to the unrolled loop nest. Then the loop header is rewritten according to the user input, and the duplicated statements are removed. The user should input the necessary information in the following format.

```
do j=1,N
  do k=1,N
    do i=1,N,B
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
      if(i+1>N) exit
      c((i+1),j)=c((i+1),j)+a((i+1),k)*b(k,j)
      if(i+2>N) exit
      c((i+2),j)=c((i+2),j)+a((i+2),k)*b(k,j)
      ...
      if(i+B-1>N) exit
      c((i+B-1),j)=c((i+B-1),j)+a((i+B-1),k)*b(k,j)
    end do
  end do
end do
```

Figure 2. A do-loop unrolled B times by *unroll loop* refactoring in Photran.

```
!$Undo unroll(factor)
```

!$Undo unroll specifies the refactoring that should be performed, and argument *factor* indicates the unroll factor that was applied to the loop nest. The new step value will be the current step value (*B* for example in Figure 2) divided by *factor*. As for the duplicated statements removal, it is assumed that the number of statements is a multiple of the unroll factor. The number of statements remaining after undoing loop unrolling, called *NoofStat*, is *the current number of statements* divided by *factor*. The first *NoofStat* statements in the loop body will be kept, and the rest statements will be removed.

With the help of our refactoring tool, the platform-specific optimization, e. g., loop unrolling applied to the code in Figure 2, is undone. Figure 3 shows the resulting code by applying the undo mechanism to the example shown in Figure 2. In this case, the loop variable is set to increment by 1 and the duplicated loop body is removed.

```
do j=1,N
  do k=1,N
    do i=1,N
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    end do
  end do
end do
```

Figure 3. The resulting do-loop after applying undo mechanism.

2) Undo Loop Tiling: Loop tiling partitions a loop's iteration into smaller chunks or blocks, thus helps eliminate as many cache misses as possible, and maximize data reuse [28].

The *tile loop* refactoring in Photran takes a double-nested do-loop of unit-step loops, and creates a nested do-loop with four levels of depth. This refactoring requires a user to provide inputs of the tile size and the tile offset, so that the user can set those parameters according to her/his will. The tile size determines the size of the accessing block. For instance, if the tile size is 3, an array will be accessed in 3×3 blocks. The tile offset adjusts where the blocks start.

Let *loopBound* and *newBound* represent the loop bounds of the double-nested do-loop before and after loop tiling, respectively. The new bounds are computed using Equation (1).

$$newBound = floor((loopBound - tileOffset) / tileSize) * tileSize + tileOffset \quad (1)$$

Figure 4 shows a typical do-loop format before loop tiling. Given a tile size "*IS*" and tile offset "*offset*," the *tile loop* refactoring in Photran always generates a tiled loop with the following format shown in Figure 5. The outer loop goes over the "blocks" and the inner loop traverses each block in its turn. Block size *IS* should be chose to fit in the cache or a memory page (whichever is smaller).

```
!before loop tiling
do j=1,N
  do i=1,N
    a(i,j)=a(i,j)+b(i,j)*a(i,j)
  end do
end do
```

Figure 4. A typical do-loop before loop tiling.

```
!after loop tiling
do j1=j_newLb, j_newUb, IS
  do i1=i_newLb, i_newUb, IS
    do j=max(j1,1), min(N, j1+IS-1)
      do i=max(i1,1), min(N, i1+IS-1)
       a(i,j)=a(i,j)+b(i,j)*a(i,j)
      end do
    end do
  end do
end do
```

Figure 5. A do-loop tiled by *tile loop* refactoring in Photran with tile size *IS*.

To perform the undo mechanism of loop tiling to a do-loop nest whose loop header has the format as shown in Figure 5, users are required to specify the do-loop nest that loop tiling was applied to and give the loop bounds for the new do-loop nest. Users should provide the necessary information in the following format prior to the do-loop nest.

```
!$Undo tile(loopName1,start1, end1,
  loopName2, start2, end2)
```

!$Undo tile specifies the refactoring that is going to be performed, and arguments *(loopName1,start1, end1, loopName2, start2, end2)* specify the new loop index variables and corresponding loop bounds. When the loop index matches the information from users and its step value does not equal to 1, its loop header is replaced as the users specified. The loop header whose index does not match the user information is removed.

With the help of our refactoring tool, the nested do-loop "after loop tiling" can be restored to that "before loop tiling." In this way, the proposed method undoes the loop tiling that has been applied to the program. Hence, the modified program can be tunable for further optimization by using auto-tuning techniques developed for un-optimized codes.

C. *Redo Mechanism of Proposed Method*

The redo mechanism is supported by an auto-tuning technique. This paper assumes auto-tuning based on full parameter search that tests all code variants of an annotated code fragment and selects the best one with the highest performance, even though any auto-tuning tools, such as the ROSE auto-tuning mechanism [29] can be employed in the proposed method. An annotation-based code generator, such as HMPPCG [30], is assumed to generate the code variants.

Generally, an auto-tuning tool is designed for un-optimized codes. Our refactoring tool, which undoes platform-specific optimizations, can refactor the *Original Program* to be tunable, thus the *Refactored Program* can easily incorporate the auto-tuning tool to obtain the best parameters for each platform. As a result, *Autotuned Program* becomes adaptable to each platform. By combining code refactoring and auto-tuning, an existing HPC application can become adaptable to different platforms. Accordingly, its performance portability is improved in a systematic way.

IV. Performance Evaluation

This section shows the evaluation of our refactoring tool and illustrates the benefit of the proposed method. The undo mechanism is supported by a code refactoring tool, which is developed by considering the Eclipse IDE [10] in mind so that the IDE can provide an interactive user interface to ask the user about necessary information for HPC refactoring. The redo mechanism is supported by an empirical auto-tuning technique that can search for the optimal tuning parameter for each different platform.

The performance of *Original Program* on the specific CPU platform, for which the program has originally been optimized, is taken as the baseline performance on each platform. We apply the proposed undo mechanism to *Original Program* and obtain *Refactored Program* whose kernel code is tunable. An auto-tuning technique based on full parameter search is applied to *Refactored Program*, and thus we can get *Autotuned Program* that has been adapted to another platform by using the auto-tuning technique. Performance portability is discussed according to the evaluation results.

A. *Experimental Setup*

To validate the effectiveness of the proposed method, we measure the execution performance of a program on different platforms to evaluate the performance portability. Platforms with different cache sizes used in the following evaluation are listed in Table I.

To show the effects of optimizations more clearly, we used the "-O0" option for the GNU compiler to disable compiler's optimizations. For the programs running on the SX-9 system, we used the "-O nounroll" option to disable the automatic unrolling optimization of the FORTRAN90/SX compiler. To evaluate the effects of HPC refactoring of unrolled loops, Fortran matrix multiplication programs of a simple triple-nested loop are used for multiplying two matrices of 512×512. With the consideration of fitting the accessed array elements into last level cache, we change the matrix size to be 3500×3500 to evaluate HPC refactoring of tiled loops.

B. *Results and Discussions*

Auto-tuning based on full parameter search is used to find the optimal unroll factor and tile size for each target platform. The execution time of the original program whose unroll factor and tile size are optimized for Intel Core i7 930 are evaluated on the four platforms listed in Table I. As shown in Figure 6, the execution time changes with the unroll factor and tile size on each platform. The experimental results indicate that the unroll factor and tile size have a considerable impact on performance of different platforms. Thus, the optimal parameters can be different for individual platforms with different configurations. It is necessary to tune those parameters to achieve high performance on each target platform. Therefore, auto-tuning is needed to achieve high performance portability. The experiment based on empirical auto-tuning found that the optimal unroll factors for platforms 1 to 4 are 64, 64, 16 and 1, respectively. The optimal tile sizes for platforms 1 to 4 are 1201, 1280, 720 and 256, respectively. These optimal parameters are used for further experiments to evaluate the performance portability.

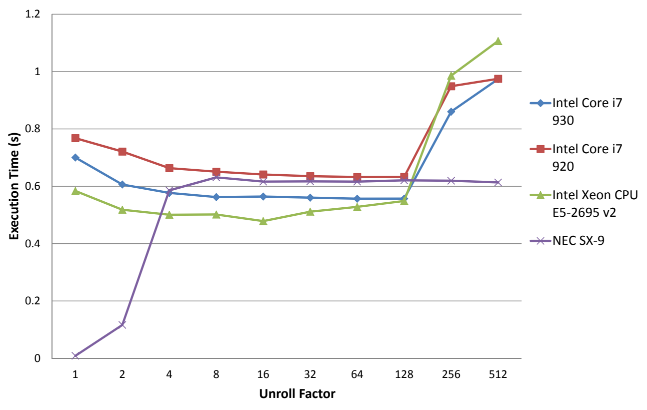
It is shown that, the performance of the SX-9 system is sensitive to the parameter configuration. This is because the *Original Program* is optimized for a totally different system with Intel Core i7. As a simple matrix multiplication program of a triple-nested loop is used in the evaluation, the sustained performance of each platform is thus limited by the cache size and memory bandwidth. The SX-9 system should achieve a much higher performance than the others. The results indicate that inappropriate optimizations mismatching the system architecture could critically degrade the sustained performance, even if the theoretical performance is high. These results hence show the importance of making an HPC application adaptive to other platforms to cope with system diversity.

To validate the usability of the proposed method, we measure the speedup ratios of target programs on each platform to evaluate the performance portability. The evaluation results are shown in Figure 7. In this figure, the horizontal axis shows the four target platforms, and the vertical axis shows the speedup on each target platform. "*Original Program*" represents the program whose unroll factor and tile size are optimized for Intel Core i7 930. "*Refactored Program*" indicates the program after undoing the optimizations with our refactoring tool. "*Autotuned Program* " shows the program that is optimized for each platform with the optimal unroll factor and tile size obtained from the previous evaluation.

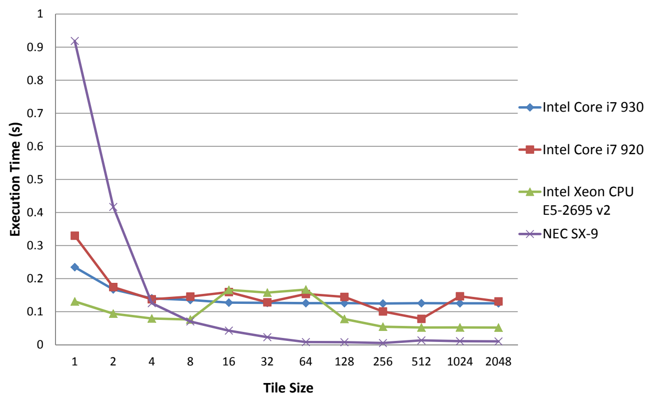
In Figure 7, *Autotuned Program* can achieve the same performance as *Original Program* on the original target platform, i .e., Intel Core i7 930. Even though *Original Program* can get fair or higher performance on platforms with high peak computational performance than Intel Core i7 930, the computational ability of each platform is not well utilized, and the performance portability of *Original Program* is low. *Refactored Program* can incorporate auto-tuning more easily than *Original Program* with the help of our refactoring tool. Thus, *Autotuned Program* can achieve comparable to or higher performance than *Original Program* on other platforms by auto-tuning optimization parameters. Accordingly, by replacing platform-specific optimizations with auto-tuning annotations, an HPC application can be performance-tunable and its performance becomes, at a certain level, portable to other platforms. These results clearly indicate the effectiveness of our method to improve the performance portability. The refactoring tools will be helpful to easily and safely do the refactoring.

TABLE I. EXPERIMENTAL ENVIRONMENT.

Platform	*CPU*	*Last Level Cache*	*Peak Computational Performance (Gflops)*	*Maximum Memory Bandwidth (GB/s)*
1	Intel Core i7 930	8MB	51.2	25.6
2	Intel Core i7 920	8MB	42.56	25.6
3	Intel Xeon CPU E5-2695 v2	30MB	230.4	59.7
4	NEC SX-9	256KB	102.4	256

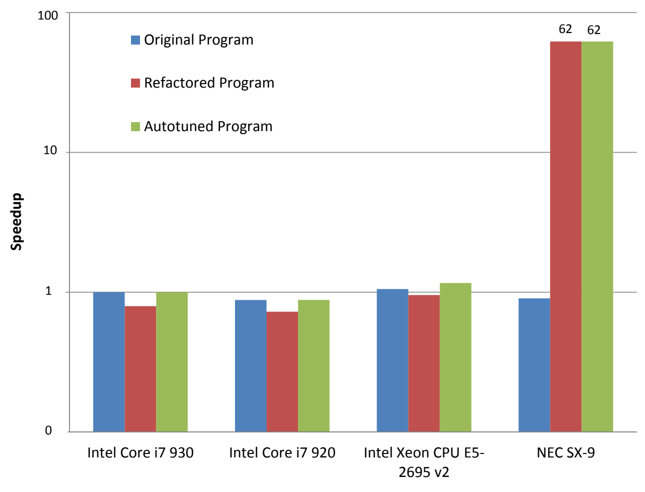


(a) Execution time changes with different unroll factors.

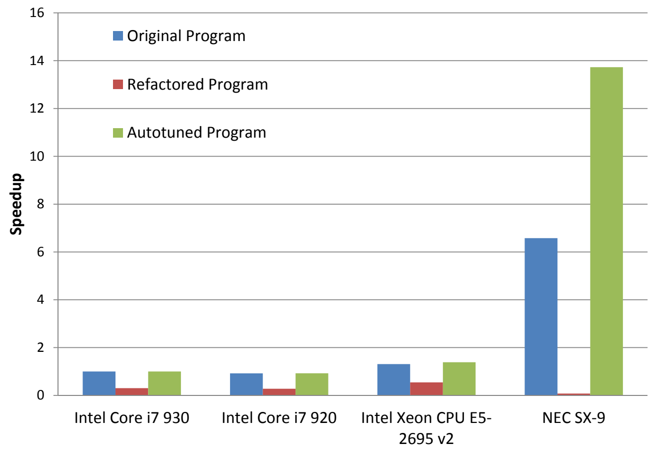


(b) Execution time changes with different tile sizes.

Figure 6. Execution performance evaluation with different parameter configurations.



(a) Speedup for Loop Unrolling.



(b) Speedup for Loop Tiling.

Figure 7. Performance evaluation results by applying the proposed method. Original Program is optimized for Intel Core i7 930.

Since platform-specific optimizations are removed from *Original Program*, *Refactored Program* runs slower than *Original Program*. The refactoring of undoing loop unrolling brings performance optimization on the NEC SX-9 platform. The vectorization processing on SX-9 is considered to be the reason of performance improvement.

The refactoring of undoing loop tiling optimization degrades the performance significantly on the NEC SX-9 platform. NEC SX-9 is equipped with an on-chip memory of 256KB called ADB (Assignable Data Buffer) [31] to realize a higher memory bandwidth as well as a shorter latency on a chip for efficient vector data accesses. The performance on SX-9 is significantly affected by the ADB size. This indicates that the appropriate optimization is crucial to the performance on different platform. The platform-specific optimizations in an application should be removed to improve the performance portability.

V. CONCLUSION AND FUTURE WORK

This paper has proposed a systematic way for improving performance portability of HPC applications by combining code refactoring and auto-tuning technologies. A semi-automated code refactoring tool that uses user knowledge is developed as an Eclipse plug-in to support undo platform-specific optimizations so that the HPC applications can be refactored to be tunable. Then, an auto-tuning technique is applied to *Refactored Program* to redo optimizations in different ways so that the application can adapt to multiple platforms. Therefore, the performance portability of an existing HPC application can be improved. The evaluation results demonstrate that combining code refactoring and auto-tuning is a promising way to replace platform-specific optimizations with auto-tuning annotations, and thereby to improve the performance portability of an existing HPC application.

This paper considered that only one kind of optimizations is applied to the selected code region. However, in practice, multiple optimizations can be applied to the same code region. In the future, we will explore more complicated codes in which multiple optimizations are applied.

ACKNOWLEDGMENT

This work was partially supported by JST CREST “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems” and Grant-in-Aid for Scientific Research (B) #25280041.

REFERENCES

[1] C. A. Mack, “Fifty years of Moore’s law,” IEEE Transactions on Semiconductor Manufacturing, vol. 24, no. 2, May 2011, pp. 202–207.

[2] J. Overbey, S. Xanthos, R. Johnson, and B. Foote, “Refactorings for Fortran and high-performance computing,” in Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications, ser. SE-HPCS’05. New York, USA: ACM, 2005, pp. 37–39.

[3] R. Johnson, B. Foote, J. Overbey, and S. Xanthos, “Changing the face of high-performance Fortran code,” White Paper, pp. 1–9, January 2006.

[4] J. Demmel, S. Williams, and K. Yelick, “Automatic performance tuning (autotuning),” in The Berkeley Par Lab: Progress in the Parallel Computing Landscape, D. Patterson, D. Gannon, and M. Wrinn, Eds. Microsoft Research, August 2013, pp. 337–339.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999, iSBN-10:0-201-48567-2.

[6] “Photran 7.0 advanced features,” The Elipse Foundation, 2011. [Online]. Available: https://wiki.eclipse.org/PTP/photran/documentation/photran7advanced [retrieved: 2015.02.09]

[7] F. G. Tinetti and M. Méndez, “Fortran legacy software: Source code update and possible parallelisation issues,” SIGPLAN Fortran Forum, vol. 31, no. 1, March 2012, pp. 5–22.

[8] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC’07), November 2007, pp. 1–12.

[9] T. Mens and T. Tourwé, “A survey of software refactoring,” IEEE Transactions on Software Engineering, vol. 30, no. 2, 2004, pp. 126–139.

[10] G. Watson, J. Alameda, B. Tibbitts, and J. Overbey, “Developing scientific applications using Eclipse and the parallel tools platform,” 2010. [Online]. Available: http://download.eclipse.org/tools/ptp/docs/ptp-sc10-tutorial.pdf [retrieved: 2015.02.09]

[11] L. Tokuda and D. Batory, “Evolving object-oriented designs with refactorings,” in Automated Software Engineering, ser. 1, vol. 8. Kluwer Academic Publishers, 2001, pp. 89–120.

[12] T. M. Peter Ebraert, Theo D’Hondt, “Enabling dynamic software evolution through automatic refactoring,” in Proceedings of the 1st Int’l Workshop on Software Evolution Transformations, November 2004, pp. 3–7.

[13] I. Moore, “Guru - a tool for automatic restructuring of self inheritance hierarchies,” in Technology of Object-Oriented Language Systems (TOOLS), vol. 17, 1995, pp. 267–275.

[14] J. L. Overbey, S. Negara, and R. E. Johnson, “Refactoring and the evolution of Fortran,” in Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, ser. SECSE’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 28–34.

[15] F. B. Kjolstad, D. Dig, and M. Snir, “Bringing the HPC programmer’s ide into the 21st century through refactoring,” in SPLASH 2010 Workshop on Concurrency for the Application Programmer (CAP’10). Association for Computing Machinery (ACM), October 2010, pp. 1–4.

[16] F. Bodin et al., “Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools,” in The second annual object-oriented numerics conference (OON-SKI, 1994, pp. 122–136.

[17] D. Orchard and A. Rice, “Upgrading Fortran source code using automatic refactoring,” in Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, ser. WRT’13. New York, USA: ACM, 2013, pp. 29–32.

[18] “SPAG - Fortran code restructuring,” Polyhedron Software Products, 2014. [Online]. Available: http://www.polyhedron.com/products/fortran-tools/plusfort-with-spag/spag-fortran-code-restructuring.html [retrieved: 2015.02.09]

[19] M. Méndez, J. Overbey, A. Garrido, F. G. Tinetti, and R. Johnson, “A catalog and classification of Fortran refactorings,” in In 11th Argentine Symposium on Software Engineering (ASSE 2010), 2010, pp. 500–505.

[20] J. Ratzinger, M. Fischer, and H. Gall, “Improving evolvability through refactoring,” SIGSOFT Softw. Eng. Notes, vol. 30, no. 4, May 2005, pp. 1–5.

[21] D. Bailey, R. Lucas, and S. Williams, Performance Tuning of Scientific Applications, ser. Chapman & Hall/CRC Computational Science. CRC Press, 2010.

[22] C. Wang, S. Hirasawa, H. Takizawa, and H. Kobayashi, “Code refactoring for high performance computing applications,” in Tohoku-Section Joint Convention Record of Institutes of Electrical and Information Engineers, 2013, p. 1A02.

[23] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,” Parallel Computing, vol. 38, no. 8, August 2012, pp. 391–407.

[24] S. Moore, “Refactoring and automated performance tuning of computational chemistry application codes,” in Simulation Conference (WSC), Proceedings of the 2012 Winter, December 2012, pp. 1–9.

[25] J. M. Levesque, R. Sankaran, and R. Grout, “Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond,” in 2012 International Conference on High Performance Computing, Networking, Storage and Analysis (SC), November 2012, pp. 1–11.

[26] M. Sugawara, K. Komatsu, S. Hirasawa, H. Takizawa, and H. Kobayashi, “Implementation and evaluation of the nanopowder growth simulation with OpenACC,” The Special Interest Group Technical Reports of IPSJ, Tech. Rep. 10, 2012.

[27] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan., “Optimal loop unrolling for GPGPU programs,” in 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), April 2010, pp. 1–11.

[28] B. Bao and C. Ding, “Defensive loop tiling for shared cache,” in 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), February 2013, pp. 1–11.

[29] C. Liao and D. Quinlan, A ROSE-Based End-to-End Empirical Tuning System for Whole Applications, Lawrence Livermore National Laboratory, Livermore, CA 94550, July 2013. [Online]. Available: http://rosecompiler.org/autoTuning.pdf [retrieved: 2015.02.09]

[30] CAPS, “HMPP codelet generator directives.” 2008. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPPCG_Directives_ReferenceManual.pdf [retrieved: 2015.02.09]

[31] T. Soga et al., “Performance evaluation of NEC SX-9 using real science and engineering applications,” in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ser. SC’09. New York, USA: ACM, 2009, pp. 28:1–28:12.

The Study of Statistical Simulation for Multicore Processor Architectures

Jongbok Lee

Dept. of Information and Communications Engineering
Hansung University
Seoul, Republic of Korea
Email: jblee@hansung.ac.kr

***Abstract*—The execution-driven or trace-driven simulation is often used for the performance analysis of widely used multicore processors in the initial design stage. However much time and disk space is necessary. In this paper, statistical simulations are performed for high performance multicore processors with various hardware configurations. For the experiment, the SPEC2000 benchmarks programs are used for the statistical profiling and synthesis. As a result, the performance obtained by the statistical simulation is comparable to that of the trace-driven simulation, with a tremendous reduction in the simulation time.**

Keywords–multicore processor, statistical simulation

I. Introduction

Currently, multicore processors are widely used for enhancing the performance of the computer system, such as smart phones, tablet PCs, notebook computers, desk top computers, etc. [1][2][3]. Since extensive simulations are necessary in the initial design stage of these multicore processors, the execution-driven simulation or trace-driven simulation is generally used. The execution-driven simulation is accurate but requires excessive time, whereas the trace-driven simulation is less accurate with the benefit of relatively reduced time. In addition, the trace-driven simulation has the disadvantage of requiring excessive disk space.

In order to address these obstacles, various alternative techniques have been studied. In the statistical simulation, the statistical characteristics of the processor architecture and the benchmark programs are collected. The statistical profiling is the collection of the characteristic distribution of the programs and the processor architectures. And then, new instruction traces are synthesized upon these statistical profiles. Since the new instruction traces are randomly generated by the statistical profiles, they represent the characteristics of each benchmark implicitly. Finally, the statistical simulation is performed with the new instruction traces on a simple statistical trace-driven simulator, which drastically reduces the simulation time [4][5][6][7][8]. Therefore, the statistical simulation can be useful for measuring the performance of multicore processors in the initial design stage, with the reduced time and space.

In this paper, the SPEC2000 integer benchmark programs are used for estimating the performance of multicore processors using statistical simulation. The result is compared with the performance of the trace-driven simulation by calculating the relative errors. This paper is organized as follows. In the Section 2, the statistical profiling will be discussed. The simulation environment will be described in Section 3. In Section 4, the simulation results will be analyzed. Finally, Section 5 concludes our paper.

II. The Statistical Profiling

The statistical simulation consists of four stages, as shown in the Figure 1.

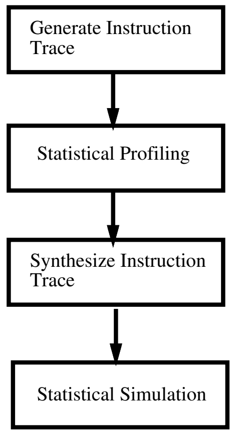


Figure 1. The statistical simulation process

The first stage is the generation of the instruction traces, which is the same as the conventional method. Each benchmark program is executed and its instruction traces are obtained by the instruction trace generator.

Secondly, the statistical profiling and the statistical data collection are performed. In this process, the instruction is analyzed and the intrinsic characteristic and the local property of each benchmark program is collected. The intrinsic characteristic of each benchmark program consists of instruction class distribution, the number of operand register distribution, and the data dependency among instructions. The instruction class distribution is obtained by reducing the original instruction set to that of only nine simple instructions. The register dependency among instructions is defined as the distance between the preceding instruction that includes the destination register and the subsequent instruction that has the source register on which it depends. These characteristics of benchmark are independent of the given microprocessor architecture, but dependent only on the instruction set and the compiler. The local characteristics include task misprediction rates, and various cache miss rates, etc. [9]. These are affected by the microprocessor hardware. This procedure is performed only once during the whole process.

Thirdly, using the collected statistical profiles of the second stage, a new benchmark is synthesized by the random number generation. A random number between 0 and 1 is generated and mapped onto the cumulative distribution function obtained from statistical profile in order to synthesize a new instruction trace.

Finally, the synthesized instruction trace is input to the statistical multicore simulator with the task prediction hit rates and the cache hit rates. Once all the data from the statistical profile and the synthetic instruction traces are secured,various experiments can be conducted by modifying the hardware architecture such as the number of cores, the task size, instruction fetch rates, the number of pipeline stages. Thus, a number of various simulations can be performed in a dramatically reduced time and space.

III. The Simulation Environment

A. The multicore processor architecture

Figure 2 shows the multicore processor with N cores. Each core is an in-order or out-of-order superscalar processor which can execute instructions in a task [2][9].

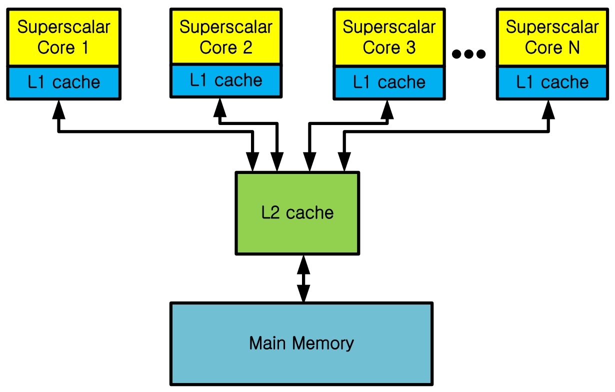


Figure 2. The multicore processor

In addition, it has a L1 instruction cache and a L1 data cache. For the cache coherency of the L1 data cache, MESI (Modified Exclusive Shared Invalidate) protocol is utilized [10]. If the data in the L1 data cache associated with a core is over-written by another core, it is invalidated. The L2 cache is shared among the cores, which is connected with the main memory.

The superscalar processor core is allocated with a task which consists of a number of instructions. The fetched instructions in the task are decoded, renamed, executed, and written back. When all the instructions in the task are retired and becomes empty, new instructions of task are fetched. If the task is mispredicted, the fetch is aborted, and all the remained instructions in the task are squashed. Since the instructions are renamed, the instructions can be issued and executed out-of-order as long as there is no true-dependency. Although the instructions can be retired out-of-order, the instructions are inserted into the reorder buffer and committed in-order as to preserve the original program order.

The detailed architecture configurations and cache parameters for each core are listed in Table I. The number of simulated cores are 2, 4, and 8. Each core is assigned one task respectively. Since the small task size cannot take the benefit of the instruction level parallelism, the task sizes are set to 4, 8, and 16. The functional unit of each core consists of a number of ALUs (Arithmetic Logic Units), load/store units according to each configuration.

TABLE I. ARCHITECTURE CONFIGURATION FOR EACH CORE.

Item		Value
number of cores		2,4,8
number of task		1
task size		4,8,16
fetch rate		2,4,8
issue rate		2,4,8
retire rate		2,4,8
functional unit	integer ALU	2,4,8
	load/store	1,2,4
L1-instruction cache		64 KB, 2-way set assoc., 16 B block, 10 cycles miss penalty
L1-data cache		64 KB, 2-way set assoc., 32 B block 10 cycles miss penalty
task address cache		2K entry
task predictor		14-bit global history based 6 cycle mispred. penalty

For the memory disambiguation, load-store and store-store pairs are inhibited from the speculative execution when the effective addresses are matched, within or among the cores. The L1 instruction cache and L1 data cache for each core is 64 KB, and it is designed as 2-way set associative. This is because the data cache hit ratio can be degraded by using MESI protocol among multicores. Tasks are predicted using the Two-level Adaptive Task Prediction scheme, and the task address cache has the size of 2048 entries. Since we do not model the main memory, the hit ratio of L2 cache is assumed to be 100 %.

B. The multicore processor simulator

Figure 3 depicts how the developed simulator works. *Initialize* function initializes all the associated variables, and *Grouping*, *Create_Window*, and *Fetch_One_Instr* function fetches new instructions every cycle. The instruction fetched by *Get_Node* function is renamed at *Rename* function by receiving timestamps. After the instruction is renamed, it is inserted into the instruction window by *Insert* function. At the *Issue* function, the instruction in the window can be retired as long as the corresponding functional unit is available and its time stamp is less than or equal to the current cycle. For the multicore simulation, *Grouping* function fills an instruction into n-cores, and *Issue* function deletes instructions according to their timestamps. This process is repeated until all the fetched instructions are deleted so that all the instruction windows are empty. Then, the cores are filled again with instructions with *Grouping* function. Since the cycle is incremented for each process, the core which spends the longest cycles determines the global cycle. If the total number of executed instructions is divided by the number of cycles spent, then IPC (Instruction per Cycle) can be obtained.

The seven SPEC 2000 integer benchmark programs that are used for the input are *bzip2,crafty, gap, gcc, gzip, parser*,

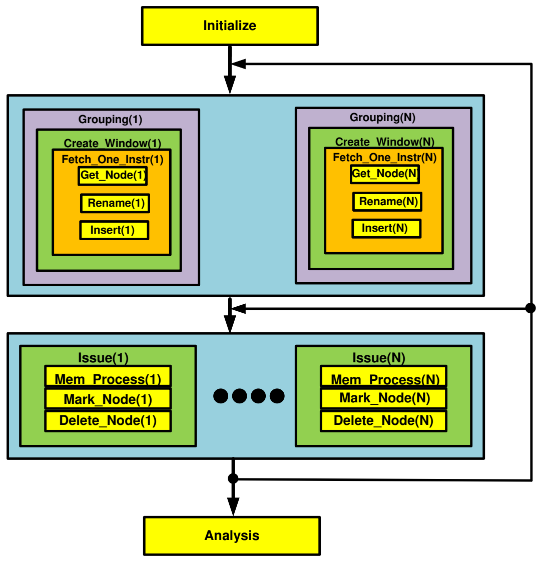


Figure 3. The flow chart of the multicore processor simulator

and *twolf*, as shown in Table II. The programs are compiled by SimpleScalar cross C compiler to obtain executables under Linux 3.3.4 [11]. The execution files are run with SimpleScalar to obtain 100 million MIPS IV instruction traces. While these are used as input for the multicore processors, the task-level parallelism is mapped onto each core.

TABLE II. SPEC 2000 BENCHMARK PROGRAMS

benchmark	description
bzip2	compression
crafty	chess game
gap	group theory interpreter
gcc	C programming language compiler
gzip	compression
parser	word processor
twolf	placement and global routing

IV. The Simulation Results

Figure 4 presents the simulation results of running SPEC 2000 integer programs on the three different task sizes of dual core, quad core, and octa core processors. The performance results obtained by the general trace-driven simulation and the statistical simulation are compared in parallel. Figure 4a and 4b are the result of the multicore processors with the maximum task size of four. For the dual core processors, the trace-driven simulation brings the geometrical mean of 1.1 IPC, whereas the statistical simulation results in 1.3 IPC. For the quad core processors, the trace-driven simulation and the statistical simulation results in 2.0 IPC and 2.4 IPC, respectively. Finally, the respective performances for the octa core processors are 3.2 IPC and 4.3 IPC. Unlike the respective relative error of 18 % and 20 % of dual core and quad core processors, the octa core processors scores the relative error of 25 % in the average.

Figure 4c and 4d show the results with the maximum task size of eight. The performance of the dual core, the quad core, and the octa core processors measured by the trace-driven simulations are 1.3 IPC, 2.4 IPC, and 4.0 IPC, respectively. The corresponding values by the statistical simulations are 1.8 IPC, 3.1 IPC, and 5.4 IPC. The average relative error results in 35 %.

Finally, Figure 4e and 4f presents the comparison result when the maximum task size is sixteen. For the dual core processors, the trace-driven simulation brings 1.5 IPC, whereas the statistical simulation results in 2.1 IPC. For the quad cores, the respective values are 2.8 IPC and 3.7 IPC. And for the octa cores, the relative error is increased by the results of 4.7 IPC and 6.4 IPC, respectively. However, the average relative error does not exceed 35 %.

As the result shows, the performance of multicore processors evaluated by statistical simulation has the similar tendency of the trace-driven simulation with the average relative error of 18 % to 35 %. Encouragingly, the average statistical simulation time is 9 seconds per benchmark program, which is 30 times faster than the trace-driven simulation. Therefore, this compensates for not being highly accurate.

V. Conclusions

In this paper, the performance of multicore processors has been evaluated and analyzed by the statistical simulation. Since the statistical simulation takes advantage of the newly synthesized instruction traces by statistical profile, the disk space is saved and the average simulation time is drastically reduced. Although the experiment shows the average relative error of 18 % to 35 %, the simulation time has been drastically reduced to 1/30.

For future research, we will study the method to further improve the accuracy of the statistical simulation, as well as expanding our scope to the multicore embedded and multicore digital signal processor architectures.

Acknowledgement

The author would like to thank Hansung University for the financial support of this research.

References

[1] T. Ungerer, B. Robic, and J. Silk, “Multithreaded Processors,” The Computer Journal, vol. 45, no. 3, 2002.

[2] S. W. Keckler, K. Olukotun, and H. P. Hofsee, Multicore Processors and Systems. Springer, 2009.

[3] M. Monchiero, “How to simulate 1000 cores,” ACM SIGARCH Computer Architecture News archive, vol. 37, no. 2, May 2009, pp. 10–19.

[4] D. B. Noonburg and J. P. Shen, “A Framework for Statistical Modeling of Superscalar Processor Performance,” in Proceedings on Third International Symposium on High Performance Computer Architecture, 1997.

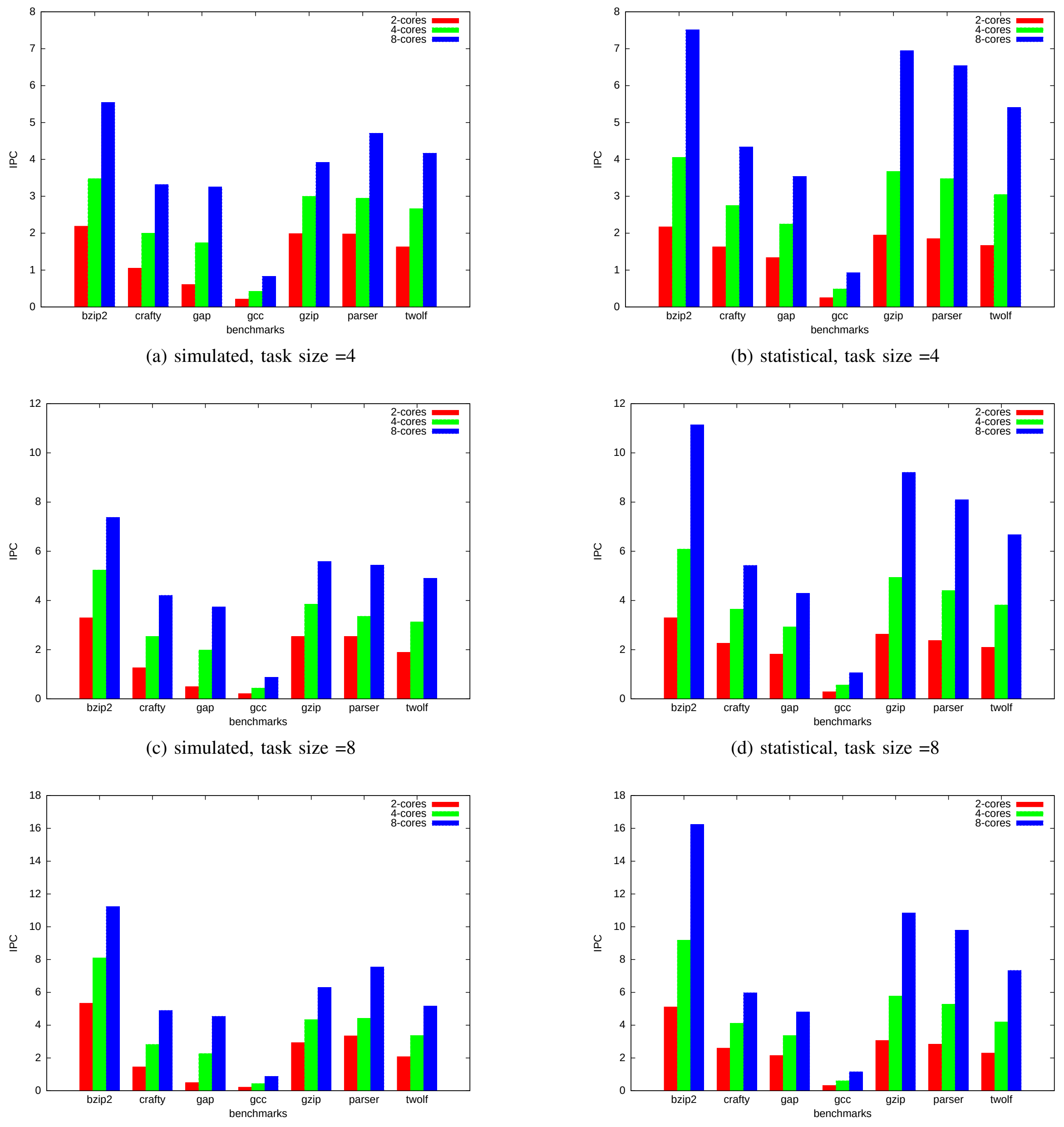
[5] R. Carl and J. E. Smith, “Modeling Superscalar Processors via Statistical Simulation,” in Workshop on Performance Analysis and Its Impact on Design, Jun. 1998.

[6] L. Eeckout, K. D. Bosschere, and H. Neefs, “Performance Analysis through Synthetic Trace Generation,” in International Symposium on Performance Analysis of Systems and Software, Apr. 2000.

[7] D. Genbrugge and L. Eckhout, “Chip multiprocessor design space exploration through statistical simulation,” IEEE Transactions on Computers, vol. 58, no. 12, Dec. 2009, pp. 1668–1681.

[8] A.Rico, A. Duran, F. Cabarcas, A. Ramirex, and M. Valero, “Trace-driven simulation of multithreaded applications,” in ISPASS, 2011.

[9] T. N. Vijaykumar and G. S. Sohi, “Task selection for a multiscalar processor,” in 31st International Symposium on Microarchitecture, Dec 1998.



(a) simulated, task size =4

(b) statistical, task size =4

(c) simulated, task size =8

(d) statistical, task size =8

(e) simulated, task size =16

(f) statistical, task size =16

Figure 4. Performance results of the trace-driven and the statistical simulation

[10] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in Proceedings of the 11th Annual International Symposium on Computer Architecture, Jun 1984, pp. 348–354.

[11] T. Austin, E. Larson, and D. Ernest, “SimpleScalar : An Infrastructure for Computer System Modeling,” Computer, vol. 35, no. 2, Feb. 2002, pp. 59–67.

A New Refutation Calculus With Logical Optimizations for PLTL

Mauro Ferrari
DiSTA
Università degli Studi dell'Insubria
Email: mauro.ferrari@uninsubria.it

Camillo Fiorentini
DI
Università degli Studi di Milano
Email: fiorentini@di.unimi.it

Guido Fiorino
DISCo
Università degli Studi di Milano-Bicocca
Email: guido.fiorino@unimib.it

***Abstract*—Propositional Linear Temporal Logic (PLTL) is a tool for reasoning about systems whose states change in time. We present an ongoing work on a new proof-search procedure for Propositional Linear Temporal Logic and its implementation. The proof-search procedure is based on a one-pass tableau calculus with a multiple-conclusion rule treating temporal-operators and on some logical optimization rules. These rules have been devised by applying techniques developed by the authors for logics with Kripke semantics and here applied to the Kripke-based semantics for Propositional Linear Temporal Logic.**

Keywords–Propositional Linear Temporal Logic; Tableaux; Satisfiability checking.

I. INTRODUCTION

In recent years, while studying proof-search procedures for non classical logics, we have introduced new tableau calculi and logical optimization rules for propositional Intuitionistic Logic (INT) [1] and propositional Gödel-Dummett Logic (DUM) [2]. As an application of these results, we have implemented theorem provers for these logics [2][3] that outperform their competitors. The above quoted calculi and optimizations are the result of a deep analysis of the Kripke semantics of the logic at hand. In this paper, we show how such semantical analysis can also be fruitfully applied to other non-classical logics with a Kripke-based semantics, by analyzing the case of PLTL. In particular, we present a new refutation tableau calculus and logical optimizations for PLTL and we briefly discuss a prototype Prolog implementation of the resulting proof-search procedure.

As for related works, our tableau calculus for PLTL lies in the line of the one-pass calculi based on sequents and tableaux of [4][5][6], whose features are suitable for automated deduction. We also cite as related the approaches based on sequent calculi discussed in [7][8] and the natural deduction based proof-search technique discussed in [9]. The results in [10][11] are based on resolution, thus they are related less to our approach.

The paper is organized as follows. Section II provides the core of our logical characterization for PLTL, Section III describes some optimization rules based on replacement of formulas, Section IV gives an account of the performances of the Prolog prototype under development, finally Section V summarizes the work with a short discussion.

II. A NEW LOGICAL CHARACTERIZATION OF PLTL

We restrict ourselves to the temporal connectives Until $\mathcal{U}$ and Next $\circ$. Entering more in details, our first contribution is a new logical characterization of PLTL by means of a tableau calculus whose distinguished feature is the multiple-conclusion rule Lin, that is a rule whose number of conclusions depends on the number of $\mathcal{U}$-formulas in the premise. The rule Lin is inspired by the multiple-conclusion rules we have developed in [2][12] to logically characterize the logic DUM. As a matter of fact, PLTL and DUM are semantically characterized by Kripke models based on linearly ordered states (we recall that PLTL and DUM have different languages and a different interpretation of the connective $\rightarrow$).

We give an account of rule Lin by means of an example which also introduces the argument to prove its correctness. Figure 1 contains the version for the language restricted to the temporal connectives $\mathcal{U}$ and $\circ$. Let us suppose that at time t a PLTL model $\underline{K}$ satisfies the set of formulas $S = \{\circ A, \circ(B\mathcal{U}C), \circ(D\mathcal{U}E)\}$ (in formulas $t \Vdash S$), where the main connective of A is not $\mathcal{U}$. This implies that: (i) $t+1 \Vdash A$ holds; (ii) there exists $t_1 > t$ such that $t_1 \Vdash C$ and, for every $t < t' < t_1$, $t' \Vdash B$ hold; (iii) there exists $t_2 > t$ such that $t_2 \Vdash E$ and, for every $t < t' < t_2$, $t' \Vdash D$ hold. The possible relationships among $t+1, t_1$ and t_2 are the following: (1) $t+1 < t_1, t_2$. In this case, $t{+}1 \Vdash \{A, B, D, \circ(B\mathcal{U}C), \circ(D\mathcal{U}E)\}$ holds; (2) $t{+}1 = t_1$ and $t_1 \leq t_2$. Hence $t+1 \Vdash \{A, C, (D\mathcal{U}E)\}$ holds; (3) $t+1 = t_2$ and $t_1 > t_2$. So we get $t+1 \Vdash \{A, B, \circ(B\mathcal{U}C), E\}$ holds. Summarizing rule Lin handles $\circ$-formulas introducing in the conclusion one branch for every formula of the kind $\circ(A\mathcal{U}B)$ and one branch for all the other kind of $\circ$-formulas. In Figure 2 we show the tableau tree for the example.

At first sight, rule Lin does not seem helpful to perform automated deduction, since it can generate an huge number of branches. However, as discussed in [2] for DUM, theorem provers using the multiple-conclusion rules as Lin can be effective. Moreover, a theorem prover can benefit from some further formulas (we call them side formulas) that we can insert in the conclusions of Lin. The side formulas represent correct information which is not necessary to handle to get the completeness. Let us suppose that neither (1) nor (2) hold. Then we can also prove that $t+1 \Vdash \neg(D \wedge \circ E)$ holds. When $D = \top$ (that is $D\mathcal{U}E$ coincides with $\diamond E$), then we can prove that for every $t' \geq t{+}2$, $t' \Vdash \neg E$ holds. This information is not necessary to the deduction but it can be exploited to perform automated deduction. In particular, when the eventuality $D\mathcal{U}E$

$$\frac{\circ A_1,\dots,\circ A_n,\circ(B_1\mathcal{U}C_1),\dots,\circ(B_m\mathcal{U}C_m)}{\mathcal{S},B_1,\dots,B_m,\circ(B_1\mathcal{U}C_1),\dots,\circ(B_m\mathcal{U}C_m)\mid\mathcal{S},C_1,B_2\mathcal{U}C_2,\dots,B_m\mathcal{U}C_m\mid H_2\mid\dots\mid H_m}\text{Lin}$$

where, $\mathcal{S}=\{A_1,\dots,A_n\}$ and for i=2,…,m

$H_i=((\mathcal{S}\cup\{\circ(B_1\mathcal{U}C_1),\dots,\circ(B_m\mathcal{U}C_m)\})\setminus\{\circ(B_i\mathcal{U}C_i),\dots,\circ(B_m\mathcal{U}C_m)\})\cup\{B_1,\dots,B_{i-1},C_i,\neg(B_i\wedge\circ C_i),B_{i+1}\mathcal{U}C_{i+1},\dots,B_m\mathcal{U}C_m\}$

Figure 1. The multiple-conclusion rule Lin

$$\cfrac{\circ A,\circ(B\mathcal{U}C),\circ(D\mathcal{U}E)}{\cfrac{A,B,D,\circ(B\mathcal{U}C),\circ(D\mathcal{U}E)}{\cfrac{C,D\mathcal{U}E}{C,E \;\Big|\; \cfrac{C,\circ(D\mathcal{U}E)}{E}\text{Lin}}\mathcal{U} \;\Big|\; \cfrac{\circ(B\mathcal{U}C),E}{C}\text{Lin}}\text{Lin} \;\Bigg|\; \cfrac{A,C,D\mathcal{U}E}{A,C,E \;\Big|\; \cfrac{A,C,\circ(D\mathcal{U}E)}{E}\text{Lin}}\mathcal{U} \;\Bigg|\; \cfrac{A,\circ(B\mathcal{U}C),E}{C}\text{Lin}}\text{Lin}$$

where $\mathcal{U}$ denotes the application of rule $\dfrac{A\mathcal{U}B}{B\mid\circ(A\mathcal{U}B)}\mathcal{U}$

Figure 2. Example of application of rule Lin

coincides with $\diamond E$ we get that from the time $t+2$ the formula E coincides with $\bot$. As a consequence, we have that if there is a loop, then t and $t+1$ are not part of the loop and a theorem prover can reset the history information. Moreover, since E is equivalent to $\bot$, it is correct to replace all the occurrences of the formula E with $\bot$. Rules based on replacement of formulas with logical constants have been proved effective both for INT [1] and for DUM [2].

III. Replacements rules

In this section, we consider $\Box$ in the language. We write $S[B/A]$ to denote the set of formulas obtained by replacing in S every occurrence of A with B. The rules Replace-$\Box$ and Replace-$\Box\neg$ given below are the analogous of rules Replace-$\mathbf{T}$ and Replace-$\mathbf{T}\neg$ given in [1][13]:

$$\frac{S,\Box A}{S[\top/A],\Box A}\text{Replace-}\Box,\quad \frac{S,\Box\neg A}{S[\bot/A],\Box\neg A}\text{Replace-}\Box\neg.$$

A special case of Replace-$\Box$ and Replace-$\Box\neg$ are the rules

$$\frac{S,A}{S\{\top/A\},A}\text{Replace-}cl,\quad \frac{S,\neg A}{S\{\bot/A\},\neg A}\text{Replace-}cl\text{-}\neg.$$

where $S\{B/A\}$ (note the curly braces) denotes the set of formulas obtained by replacing with B the occurrences of A in S that are not under the scope of any temporal connective.

In [1], we have introduced some variants of Replace-$\Box$ and Replace-$\Box\neg$ based on the Kleene sign property. We exploit some conditions under which we can replace a propositional variable p applying the rules Replace-$\Box$ and Replace-$\Box\neg$ also when neither $\Box p$ nor $\Box\neg p$ explicitly occur in the premise of the rule. The condition for the applicability of these rules is based on the notion of *polarity* of p: p can be eliminated from a set of formulas S (replaced with $\top$ or $\bot$) if all the occurrences of p in S have the same polarity. The notion of polarity can be easily explained in the framework of PLTL where formulas of the kind $A\to B$ are written as $\neg A\vee B$ and Negation Normal Form is applied: a propositional variable p occurs with positive (resp. negative) polarity in a set S, denoted with $p\preceq^+ S$ (resp. $p\preceq^- S$) iff no occurrence (resp. every occurrence) of p in S is of the kind $\neg p$. The following are replacement rules of propositional variables fulfilling the notion of positive or negative occurrence in a set that can be computed in linear time on S:

$$\frac{S}{S[\top/p]}\preceq^+,\ \text{provided } p\preceq^+ S;\quad \frac{S}{S[\bot/p]}\preceq^-,\ \text{provided } p\preceq^- S.$$

In Figure 3 we show a piece of deduction for the formula `acacia-demo-v3_1`, where also we apply some obvious boolean simplifications.

IV. Preliminary results

We have developed a Prolog prototype to perform some experiments on the benchmark formulas for PLTL. The development of the prover is at the very early stage. We have only focused on the implementation of the logical calculus with rule Lin in its first simplified version (without side formulas) and the optimization rules provided in Section III. The part of the prover related to the history construction, loop-checking and loop-satisfaction is very naive. Thus, in general, the known provers outperform our implementation. However, there are some remarks related to the proposed optimizations that deserve a comment. First, all the optimizations rules we have described are effective in speeding-up the deduction. Second, the rules $\preceq^+$ and $\preceq^-$ apply to the benchmark formulas `acacia-demo-v3`, `alaska-szymanski`, `rozier` (some subfamilies), `O1-schuppan` and `trp` (some subfamilies) without requiring any deduction step. On our Mac OS X (2.7 GHz, Core i7, 8GB), in less than 10 (often in less that 1) seconds, the prototype decides the families `acacia`, `alaska-lift` (except for the non-negated `l` variant), `alaska-szymanski`, `anzu-amba`, `anzu-amba_c` and `anzu-amba_cl` in negated version, `forobotsr1f0` (many formulas in the negated version and in a few cases also the non-negated versions), `rozier-formulas`, `rozier-patterns`, `schuppan-O1` and `trp` (some cases). Without the described optimizations timings would be greater by some order of magnitudes.

$$\dfrac{H \equiv \top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\neg req \vee \circ grt \vee \circ\circ grt \vee \circ\circ\circ grt) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\neg\bot \vee \circ grt \vee \circ\circ grt \vee \circ\circ\circ grt) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}\preceq^-,\, req\preceq^- H$$

$$\dfrac{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\neg\bot \vee \circ grt \vee \circ\circ grt \vee \circ\circ\circ grt) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top \vee \circ grt \vee \circ\circ grt \vee \circ\circ\circ grt) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}\texttt{bool}$$

$$\dfrac{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top \vee \circ grt \vee \circ\circ grt \vee \circ\circ\circ grt) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}{H' \equiv \top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}\texttt{bool}$$

$$\dfrac{H' \equiv \top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\neg grt \vee \circ\neg grt) \wedge \Box(\neg canc \vee \circ(\neg grt\,\mathcal{U}\,go)))}{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\neg\bot \vee \circ\neg\bot) \wedge \Box(\neg canc \vee \circ(\neg\bot\,\mathcal{U}\,go)))}\preceq^-,\, grt\preceq^- H'$$

$$\dfrac{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\neg\bot \vee \circ\neg\bot) \wedge \Box(\neg canc \vee \circ(\neg\bot\,\mathcal{U}\,go)))}{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\top \vee \circ\top) \wedge \Box(\neg canc \vee \circ(\top\,\mathcal{U}\,go)))}\texttt{bool}$$

$$\dfrac{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\top \vee \circ\top) \wedge \Box(\neg canc \vee \circ(\top\,\mathcal{U}\,go)))}{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\top) \wedge \Box(\neg canc \vee \circ(\top\,\mathcal{U}\,go)))}\texttt{bool}$$

$$\dfrac{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee (\Box(\top) \wedge \Box(\top) \wedge \Box(\neg canc \vee \circ(\top\,\mathcal{U}\,go)))}{\top\mathcal{U}\,(canc \wedge \circ\neg go) \vee \Box(\neg canc \vee \circ(\top\,\mathcal{U}\,go))}\texttt{bool}$$

Figure 3. Example of application of rule $\preceq^-$ to the formula `acacia-demo-v3_1`

V. Conclusion and Future Work

We have presented our ongoing research on automated deduction for PLTL. We face the problem along different lines. In this note we have discussed two of them: Section II provides some ideas for a new proof-theoretical characterization of PLTL based on a multiple-conclusion rule; Section III describes logical rules to cut the size of the proofs. In addition to the given results, an important part of the future work is to exploit the notions of *local formula* [3] and *evaluation* [14] to develop an advanced strategy to avoid some rule application.

Acknowledgment

The third author acknowledges the support of the MIUR PRIN 2010-2011 grant "Automi e Linguaggi Formali: Aspetti Matematici e Applicativi", code 2010LYA9RH.

References

[1] M. Ferrari, C. Fiorentini, and G. Fiorino, "Simplification rules for intuitionistic propositional tableaux," ACM Trans. Comput. Logic, vol. 13, no. 2, Apr. 2012, pp. 14:1–14:23. [Online]. Available: http://doi.acm.org/10.1145/2159531.2159536

[2] G. Fiorino, "Refutation in Dummett logic using a sign to express the truth at the next possible world," in IJCAI, T. Walsh, Ed. IJCAI/AAAI, 2011, pp. 869–874.

[3] M. Ferrari, C. Fiorentini, and G. Fiorino, "fCube: An efficient prover for intuitionistic propositional logic," in LPAR (Yogyakarta), ser. Lecture Notes in Computer Science, C. G. Fermüller and A. Voronkov, Eds., vol. 6397. Springer, 2010, pp. 294–301.

[4] K. Brünnler and M. Lange, "Cut-free sequent systems for temporal logic," J. Log. Algebr. Program., vol. 76, no. 2, 2008, pp. 216–225.

[5] J. Gaintzarain, M. Hermo, P. Lucio, M. Navarro, and F. Orejas, "Dual systems of tableaux and sequents for PLTL," J. Log. Algebr. Program., vol. 78, no. 8, 2009, pp. 701–722.

[6] S. Schwendimann, "A new one-pass tableau calculus for PLTL," in Tableaux'98, 1998, pp. 277–291.

[7] R. Pliuskevicius, "Investigation of finitary calculus for a discrete linear time logic by means of infinitary calculus," in Baltic Computer Science, ser. Lecture Notes in Computer Science, J. Barzdins and D. Bjørner, Eds., vol. 502. Springer, 1991, pp. 504–528.

[8] B. Paech, "Gentzen-systems for propositional temporal logics," in CSL, ser. Lecture Notes in Computer Science, E. Börger, H. K. Büning, and M. M. Richter, Eds., vol. 385. Springer, 1988, pp. 240–253.

[9] A. Bolotov, O. Grigoriev, and V. Shangin, "Automated natural deduction for propositional linear-time temporal logic," in TIME. IEEE Computer Society, 2007, pp. 47–58.

[10] M. Fisher, C. Dixon, and M. Peim, "Clausal temporal resolution," ACM Trans. Comput. Log., vol. 2, no. 1, 2001, pp. 12–56.

[11] M. Suda and C. Weidenbach, "Labelled superposition for PLTL," in LPAR, ser. Lecture Notes in Computer Science, N. Bjørner and A. Voronkov, Eds., vol. 7180. Springer, 2012, pp. 391–405.

[12] G. Fiorino, "Tableau calculus based on a multiple premise rule," Information Sciences, vol. 180, no. 19, 2010, pp. 371–399.

[13] F. Massacci, "Simplification: A general constraint propagation technique for propositional and modal tableaux," in Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Oosterwijk, The Netherlands, ser. LNCS, H. de Swart, Ed., vol. 1397. Springer-Verlag, 1998, pp. 217–232.

[14] M. Ferrari, C. Fiorentini, and G. Fiorino, "An evaluation-driven decision procedure for G3i," TOCL, in press.