An Algebraic Framework for Optimizing Parallel Programs

Ichiro Satoh
Department of Information Sciences, Ochanomizu University
2-1-1 Otsuka Bunkyo-ku Tokyo 112, Japan
ichiro@is.ocha.ac.jp

Abstract

This paper proposes a theoretical framework for verifying and deriving code optimizations for programs written in parallel programming languages. The key idea of this framework is to formalize code optimizations as compositional transformation rules for programs presented as terms of an enriched process calculus. The rules are formulated on the basis of an algebraic order relation between two programs which states that they are behaviorally equivalent and one of them is faster than the other. The correctness and effectiveness of optimized programs derived from the rules can be ensured in all circumstances. The framework is unique among other existing works in being able to quantitatively analyze the temporal costs of synchronizations among parallel programs. This paper presents basic ideas and definitions of the framework with several examples.

1. Introduction

Parallel computation will play an increasingly important role in many areas of computer systems. As it becomes popular, customers will demand more efficient execution of parallel programs. Code optimization is crucial to achieving good performance of parallel computation. A number of code optimizations have been studied and used for parallel language compilers.

These optimizations are required to be *correct* and *effective* in all circumstances, in the sense that they do not change the functional behaviors of their original programs and the optimized programs can perform the behaviors faster than the original ones. However, most existing code optimizations are described in ad hoc fashions. It is difficult to predict what the application of such optimizations will generate. A few techniques have been formally given but they are isolated from other optimization techniques. As a result, we cannot often understand whether a combination of more than one optimization is valid or not. Therefore, we need a formal and common way to analyze optimized programs of parallel programming languages.

The final goal of this work is to construct a theoretical framework for verifying and deriving various optimizations for parallel programs. As the first step in developing such a framework, this paper addresses several optimizations to minimize the overheads of synchronizations and communications among parallel processes, in particular optimizations to reduce synchronization time.[1] This is because these optimizations are very effective for parallel programs. The cost of synchronization goes far beyond just the operations that manipulate data structures in programs, and a processor may often have to wait for other processors to reach the same synchronization point. The framework presented in this paper is characterized among other existing works by having the ability to quantitatively analyze the temporal costs of synchronizations among parallel programs. On the other hand, it may not always be suitable for formulating other optimization techniques, for example redundancy elimination and lock elimination. We believe that such techniques should be analyzed by means of other methods, for example data dependence analysis and control flow analysis.

Organization of this paper:

In the next section we briefly present our basic ideas concerning the formalization of optimizations. In Section 3 we define an extended process calculus and in Section 4 formulate an algebraic order relation as justification for optimized programs. In Section 5 we formalize several optimization techniques as a rewriting

[1] We show that this framework can formalize other optimizations, for example parallelization techniques, which detect independent or commutable pieces from a program and then generate code that executes such pieces in parallel.

system that consists of transformation rules between programs and their optimized ones. In Section 6 we compare related work and in the final section we give some concluding remarks and mention future work.

2. Preliminaries

The framework presented in this paper consists of a specification language, algebraic semantics for optimized programs, and compositional optimizing transformations of programs. The goal of this paper is to provide a suitable framework for verifying and deriving optimizations to reduce the cost of synchronizations among parallel programs and to parallelize sequential programs.

Specification Language

The language is given as an extended process calculus with the following constructors. This is because process calculi provide a well-founded basis for analyzing parallel computation. The other constructions of this calculus are basically inherited from those of existing process calculi, for example π-calculus [13], CCS [11], and CSP [5].

Specifying Commutativity in Program

A program often contains commutable pieces of code, that generate the same result regardless of the order in which they execute. Such commutable code plays an important role in many optimization techniques. Therefore, we introduce a special constructor to specify commutable subprograms, written as $\otimes$. For example $P_1 \otimes P_2$ means that the order of execution is either P_1 followed by P_2 or P_2 followed by P_1, nondeterministically. This constructor is not used as an imperative constructor of programming languages. $P_1 \otimes P_2$ is just a specification that denotes P_1 and P_2 to be commutable subprograms in a sequential program. It is worth mentioning that this framework does not intend to detect such commutable pieces in a program, because they should be analyzed by other methods, for example data dependency analysis and control flow analysis.

Specifying Conditional Branch in Program

We also introduce a nondeterministic choice operator, written as $\oplus$. This, like $\otimes$, is not just a program but is intended to specify various nondeterministic properties in programs. $P_1 \oplus P_2$ behaves like either P_1 or P_2 nondeterministically. By using this operator, we thereby specify conditional branch in a program, for example an *if-then-else* statement or a *while-loop* statement whose choices cannot be predicted exactly by using other analysis methods.

Specifying Temporal Cost of Computation

In order to analyze optimizations for reducing computation time and synchronization time, the language needs to be quantitatively specify the temporal costs of computation and synchronization. We introduce a new prefix operator to express various execution costs. This suspends a program for a specified period, written as $\langle t \rangle$, where t is the amount of the suspension. For instance, $\langle t \rangle ; P$ represents a program which is idle for t time units and then behaves as P. By using this operator, we can quantitatively reflect various execution costs of real computation on the specification of the computation. Instead, we assume that all communications take no time. We also assume that a program having a possible communication action is blocked until another program becomes ready to communicate with it. This assumption enables us to predict the temporal costs of synchronization exactly. Synchronization cost is modeled as the amount of the blocking time.

Algebraic Semantics on Optimization

We formulate an algebraic order relation, called *optimization order relation*, to verify the correctness and temporal effectiveness of optimizations. It orders two programs presented as expressions of the specification language, with respect to their execution speeds, using the symbol $\gtrsim$. For example, $P_1 \gtrsim P_2$ means that program P_2 is more optimized than program P_1, in the sense that P_1 and P_2 have the same functional behaviors and P_2 can perform them faster than P_1. By using this order relation, we can prove whether an optimized program satisfies the following two properties:

correctness: the functional result of an optimized program is not different from that of its original one, and

effectiveness: the performance of the optimized program is faster than that of the original one in all circumstances.

Optimizing Transformation

A code optimization can be simplified to finding code that is behaviorally equivalent and can be executed faster than the original code. We thereby formalize code optimizations as transformations from programs into their optimized ones. The transformations are given in the form of inference rules for programs described in the calculus. The assumption of each inference rule specifies the condition under which its optimization can be applied, and its conclusion reflects the effect of applying of the optimization to a program satisfying the condition. A series of applications of the rules to a program is regarded as a process of optimizing the program in real optimizing compilers. Since the central notion of the rules is based on the optimization order relation, the correctness and effectiveness of optimized programs from the rules can be ensured in any context.

3. Specification Language

This language is formulated based on time extended process calculi studied in [20, 23]. This section defines the syntax and the semantics of the calculus.

Syntax

Definition 3.1 Let $\mathcal{T}$ be the set of time values ranged over by t,t_1,t_2,.... It has the following operation:

$$t_1 + t_2 = t_2 + t_1, \qquad t_1 + (t_2 + t_3) = (t_1 + t_2) + t_3,$$
$$t + 0 = t = 0 + t,$$
$$t_1 + t = t_2 + t \textit{ then } t_1 = t_2, \quad t_1 \leq t_2 \textit{ iff } \exists t : t_1 + t = t_2$$
$$t_1 \dot{-} t_2 = \begin{cases} t \quad \textit{where} \quad t_1 + t = t_2 & \textit{if } t_1 \leq t_2 \\ 0 & \textit{otherwise} \end{cases}$$

□

Hereafter, we assume $\mathcal{T}$ to be the set of positive natural numbers including 0. Next, we define symbols to present the events of processes.

Definition 3.2

- Let $\mathcal{A}$ be an infinite set of names denoting communication actions. Its elements are denoted as a,b,...
- Let $\overline{\mathcal{A}}$ be an infinite set of co-names. Its elements are denoted as $\overline{a}$,$\overline{b}$,..., where $\overline{a}$ is the complementary action of a.
- Let $\mathcal{L} \equiv \mathcal{A} \cup \overline{\mathcal{A}}$ be a set of communication action names. Elements of the set are written as $\ell, \ell', \ldots$.
- Let τ denote a handshake communication. It is considered to be unobservable from outside environments.
- Let $\surd$ denote a program termination.
- Let $Act \equiv \mathcal{L} \cup \{\tau\}$ be the set of operational actions. Its elements are denoted as α,β,.... □

Definition 3.3 The set $\mathcal{P}$ of expressions of the calculus, ranged over by $P, P_1, P_2, \ldots$ is defined recursively by the following abstract syntax:

$$\begin{array}{rcl} P & ::= & \mathbf{0} \ \mid \ (\pi) \ \mid \ \langle t \rangle \ \mid \ P_1 \,;\, P_2 \ \mid \ P_1 \parallel P_2 \\ & & \mid \ P_1 \oplus P_2 \ \mid \ P_1 \otimes P_2 \ \mid \ A \\ \pi & ::= & \ell \ \mid \ \pi_1 | \pi_2 \end{array}$$

where t is an element of $\mathcal{T}$. A is a process variable and is always used in the form $A \overset{def}{=} P$. We assume that in $A \overset{def}{=} P$, P is always closed, and each occurrence of A in P is only within some subexpressions, $(\pi); A$ where π is not empty. In $(\ell_1|\ldots|\ell_n)$, $\ell_1, \ldots, \ell_n$ are distinct and we denote (π) as $()$ when π is empty. When P is of the form $P_1 \parallel P_2$, $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$, where $\mathcal{L}(P)$ is inductively defined by: $\mathcal{L}(\surd) = \mathcal{L}(\langle t \rangle) = \emptyset$, $\mathcal{L}((\ell_1, \ldots, \ell_n)) = \{\ell_1, \ldots, \ell_n\}$, $\mathcal{L}(P_1; P_2) = \mathcal{L}(P_1 \parallel P_2)$ $= \mathcal{L}(P_1 \oplus P_2) = \mathcal{L}(P_1 \otimes P_2) = \mathcal{L}(P_1) \cup \mathcal{L}(P_2)$. In $A \overset{def}{=} P$, we have $\mathcal{L}(P) \subseteq \mathcal{L}(A)$. We often abbreviate (ℓ) to ℓ. We describe an empty element of $\mathcal{L}(P)$ as ϵ. □

The intuitive meanings of some basic expressions in the calculus are as follows:

- $\mathbf{0}$ represents a terminated program.
- $(\ell_1|\ell_2); P$ represents that a process performs either ℓ_1 followed by ℓ_2 and then P, or ℓ_2 followed by ℓ_2 and then P.
- $P_1 \,;\, P_2$ denotes the sequential composition of two program P_1 and P_2. If the execution of P_1 terminates then the execution of P_2 follows that of P_1.
- $P_1 \parallel P_2$ expresses that P_1 and P_2 can execute in parallel. Note that P_1 does not contain all the communication actions included in P_2, and vice versa.
- $A \overset{def}{=} P$ has a process identifier A and occurrences of A in expressions are interpreted as P.
- $P_1 \oplus P_2$ behaves like either P_1 or P_2 nondeterministically.
- $P_1 \otimes P_2$ is either P_1 followed by P_2 or P_2 followed by P_1 nondeterministically. It shows that P_1 and P_2 are commutable programs.
- $\langle t \rangle$ is idle for t time units. For example, $\langle t \rangle; P$ behaves like P after t time units. It is used for quantitatively specifying the execution time of an internal processing.

All communications in the calculus are assumed to be one-way synchronous communication. A program having a communication action must wait until another program becomes ready to communicate with it. Once both the partners of a communication are ready, they must perform the actions immediately. It is worth mentioning that the calculus is designed as just a specification language for describing code optimizations.

Operational Semantics

The operational semantics of the calculus is defined as two layers of labeled transition rules: *behavioral transition*, written as $\xrightarrow{\mu}$ ($\longrightarrow \subseteq \mathcal{P} \times (\mathcal{A} \cup \{\surd\}) \times \mathcal{P}$), and *temporal transition*, written as $\xrightarrow{\langle 1\rangle}$ ($\longrightarrow \subseteq \mathcal{P} \times \{\langle 1\rangle\} \times \mathcal{P}$). The former transition defines the semantics of functional behaviors of programs. For example $P \xrightarrow{\mu} P'$ means that P performs action μ and then behaves as P'. The latter transition corresponds to the advance of time. For example, $P \xrightarrow{\langle 1\rangle} P'$ means that process P becomes P' after one time unit. We assume that time passes at the same speed in all processes and all communication and internal actions are instantaneous. The definition of the semantics also uses a structural congruence ($\equiv$) over expressions in $\mathcal{P}$ as studied in [12].

Definition 3.4 $\equiv$ is defined as follows:

(1) $P_1;(P_2;P_3) \equiv (P_1;P_2);P_3 \qquad \mathbf{0};P \equiv P$

(2) $P_1 \parallel P_2 \equiv P_2 \parallel P_1 \qquad P \parallel \mathbf{0} \equiv P$
$P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3$

(3) $P_1 \oplus P_2 \equiv P_2 \oplus P_1 \quad P_1 \oplus (P_2 \oplus P_3) \equiv (P_1 \oplus P_2) \oplus P_3$
$P \oplus P \equiv P$

(4) $P_1 \otimes P_2 \equiv (P_1;P_2) \oplus (P_2;P_1)$

(5) $A \equiv P$ if $A \overset{def}{=} P$

(6) $() \equiv \mathbf{0} \quad \pi_1|\pi_2 \equiv \pi_2|\pi_1 \quad \pi_1|(\pi_2|\pi_3) \equiv (\pi_1|\pi_2)|\pi_3$

(7) $\langle 0\rangle \equiv \mathbf{0}$

where we assume α-equivalence and syntactic equivalence are included in $\equiv$. We abbreviate transitive closure of $\equiv$ to $\equiv$. □

Definition 3.5 The calculus is a labeled transition system $\langle\ \mathcal{P}, \mathcal{A} \cup \{\langle 1\rangle\} \cup \{\surd\}, \{\xrightarrow{\mu} \subseteq \mathcal{P} \times \mathcal{P} \,|\, \mu \in \mathcal{A} \cup \{\langle 1\rangle\} \cup \{\surd\}\ \}\ \rangle$. The transition relation $\longrightarrow$ is defined by two kinds of structural induction rules as given below:

$$\frac{-}{(\ell_1|\cdots|\ell_i|\cdots|\ell_n) \xrightarrow{\ell_i} (\ell_1|\cdots|\ell_{i-1}|\ell_{i+1}|\cdots|\ell_n)}$$

$$\frac{-}{\mathbf{0} \xrightarrow{\surd} \epsilon} \qquad \frac{-}{P_1 \oplus P_2 \xrightarrow{\tau} P_1} \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1\,;\,P_2 \xrightarrow{\alpha} P_1'\,;\,P_2}$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \parallel P_2 \xrightarrow{\alpha} P_1' \parallel P_2} \qquad \frac{P_1 \xrightarrow{\overline{\ell}} P_1' \quad P_2 \xrightarrow{\ell} P_2'}{P_1 \parallel P_2 \xrightarrow{\tau} P_1' \parallel P_2'}$$

$$\frac{P_1 \equiv P_2 \quad P_1 \xrightarrow{\alpha} P_1' \quad P_1' \equiv P_2'}{P_2 \xrightarrow{\alpha} P_2'}$$

$$\frac{-}{(\pi) \xrightarrow{\langle 1\rangle} (\pi)} \qquad \frac{-}{\langle t+1\rangle \xrightarrow{\langle 1\rangle} \langle t\rangle}$$

$$\frac{P_1 \xrightarrow{\langle 1\rangle} P_1'}{P_1\,;\,P_2 \xrightarrow{\langle 1\rangle} P_1'\,;\,P_2}(P_1 \not\equiv \mathbf{0})$$

$$\frac{P_1 \xrightarrow{\langle 1\rangle} P_1' \quad P_2 \xrightarrow{\langle 1\rangle} P_2'}{P_1 \parallel P_2 \xrightarrow{\langle 1\rangle} P_1' \parallel P_2'}\ (\neg\exists P : P_1 \parallel P_2 \xrightarrow{\tau} P)$$

$$\frac{P_1 \equiv P_2 \quad P_1 \xrightarrow{\langle 1\rangle} P_1' \quad P_1' \equiv P_2'}{P_2 \xrightarrow{\langle 1\rangle} P_2'}$$

where $\alpha \in Act$ and $\ell \in \mathcal{L}$. π is not empty. ϵ is an empty symbol of $\mathcal{P}$ and α is an action in $\mathcal{A}$. □

Note that $\surd$ is performed only at the termination of all parallel processes since α is not $\surd$ and $P_1;P_2$, $P_1 \parallel P_2$, and $P_1 \oplus P_2$ do not have any $\surd$ transition.

Example 3.6

(1) $a;\langle 2\rangle;\overline{b}$ represents a process which after receiving input action a is suspended for two time units and then can send output action $\overline{b}$.

(2) $(a|b|c);P_1$ is a process which has three possible input actions: a, b, and c. Even if it can perform the actions in any sequential order, it becomes P_1.

(3) $\overline{a};\overline{b};P_2$ specifies a process which can perform output action $\overline{a}$ and then behaves as $\overline{b};P_2$.

(4) $(a|b|c);P_1 \parallel \overline{a};\overline{b};P_2$ means that $(a|b|c);P_1$ and $\overline{a};\overline{b};P_2$ can execute in parallel.

(5) $a;(\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle))$ expresses a process which after receiving input action a, behaves as either $\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle$ or $\langle 2\rangle;\overline{b};\langle 1\rangle;\langle 4\rangle$.

The transition relations given in Definition 3.5 do not distinguish between observable actions and unobservable ones. We define some transition relations due to the non-observability of τ.

Definition 3.7 Let $\mu \in \mathcal{A} \cup \{\surd\}$ and $t, t_1, \in \mathcal{T}$ then

(1) $\overset{\mu}{\Longrightarrow}$ is defined as $(\xrightarrow{\tau})^*\xrightarrow{\mu}(\xrightarrow{\tau})^*$.

(2) $\overset{\widehat{\mu}}{\Longrightarrow}$ is defined as $\overset{\mu}{\Longrightarrow}$ if $\mu \neq \tau$, otherwise $(\xrightarrow{\tau})^*$.

(3) $\xrightarrow{\langle t\rangle}$ is defined as $(\xrightarrow{\langle 1\rangle})^t$.

(4) $\overset{\langle t\rangle}{\Longrightarrow}$
is defined as $(\xrightarrow{\tau})^*\xrightarrow{\langle t_1\rangle}(\xrightarrow{\tau})^*\cdots(\xrightarrow{\tau})^*\xrightarrow{\langle t_n\rangle}(\xrightarrow{\tau})^*$ where $t = t_1 + \cdots + t_{n-1}$. □

We study some essential properties of the semantics of the calculus.

Proposition 3.8 Let $P \in \mathcal{P}$. Then

(1) if $P \xrightarrow{\tau} P'$ then $\neg\exists P''$: $P \xrightarrow{\langle 1\rangle} P''$

(2) $P \xrightarrow{\langle 1\rangle} P'$ and $P \xrightarrow{\langle 1\rangle} P''$ then $P' \equiv P''$ □

The first property means that if a process has an executable communication or an internal action, it must perform the action immediately without imposing unnecessary idling. This assumption is the same as the notion of *maximal progress* shown in [20, 28]. It lets us measure the minimum cost in synchronization among parallel processes. The second means that time is deterministic in the sense that the passage of time does not interfere with any non-determinism.

Example 3.9 We show partial transitions of some basic expressions.

(1) We show a partial transition of communicating processes.

$$\begin{aligned}(a|b|c);P_1 \parallel \overline{a};\overline{b};P_2 &\xrightarrow{\tau} (b|c);P_1 \parallel \overline{b};P_2\\ &\xrightarrow{\tau} (c);P_1 \parallel P_2\end{aligned}$$

Moreover, we have

$$\begin{aligned}(a|b|c);P_1 \parallel \overline{b};\overline{a};P_2 &\xrightarrow{\tau} (a|c);P_1 \parallel \overline{a};P_2\\ &\xrightarrow{\tau} (c);P_1 \parallel P_2\end{aligned}$$

(2) $a;(\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle))$ represents a specification of the program which is allowed to behave like either $\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle$ or $\langle 2\rangle;\overline{b};\langle 1\rangle;\langle 4\rangle$ after receiving action a. The former specifies a program which after performing an internal computation for four and two units of time, sends action b and then performs an internal computation for one unit of time. The latter sends action b after two time units and then performs an internal computation for one and four time units. We show a partial transition of this expression.

$$\begin{aligned}a;(\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle)) &\xrightarrow{a} \langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle)\\ &\xrightarrow{\tau} \langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle\\ &\qquad\text{or}\quad \langle 2\rangle;\overline{b};\langle 1\rangle;\langle 4\rangle\end{aligned}$$

Consider the case of $\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle$.

$$\begin{aligned}\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle &\xrightarrow{\langle 6\rangle} \overline{b};\langle 1\rangle\\ &\xrightarrow{\overline{b}} \langle 1\rangle\\ &\xrightarrow{\langle 1\rangle} \mathbf{0}\\ &\xrightarrow{\surd} \epsilon\end{aligned}$$

(3) Consider an interaction between two parallel programs: $(\langle 1\rangle;\overline{a};\langle 3\rangle;b) \parallel (a;(\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle)))$. It is described by the following transition:

$$\begin{aligned}&\langle 1\rangle;\overline{a};\langle 3\rangle;b \parallel a;(\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle))\\ &\xrightarrow{\langle 1\rangle} \overline{a};\langle 3\rangle;b \parallel a;(\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle))\\ &\xrightarrow{\tau} \langle 3\rangle;b \parallel (\langle 4\rangle \otimes (\langle 2\rangle;\overline{b};\langle 1\rangle))\\ &\xrightarrow{\tau} \langle 3\rangle;b \parallel (\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle)\\ &\qquad\text{or}\quad \langle 3\rangle;b \parallel (\langle 2\rangle;\overline{b};\langle 1\rangle;\langle 4\rangle)\end{aligned}$$

Case: $\langle 3\rangle;b \parallel (\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle)$

$$\begin{aligned}\langle 3\rangle;b \parallel (\langle 4\rangle;\langle 2\rangle;\overline{b};\langle 1\rangle) &\xrightarrow{\langle 6\rangle} b \parallel (\overline{b};\langle 1\rangle)\\ &\xrightarrow{\tau} \langle 1\rangle\\ &\xrightarrow{\langle 1\rangle} \mathbf{0}\\ &\xrightarrow{\surd} \epsilon\end{aligned}$$

Case: $\langle 3\rangle;b \parallel (\langle 2\rangle;\overline{b};\langle 1\rangle;\langle 4\rangle)$

$$\begin{aligned}\langle 3\rangle;b \parallel (\langle 2\rangle;\overline{b};\langle 1\rangle;\langle 4\rangle) &\xrightarrow{\langle 3\rangle} b \parallel (\overline{b};\langle 1\rangle;\langle 4\rangle)\\ &\xrightarrow{\tau} \langle 1\rangle;\langle 4\rangle\\ &\xrightarrow{\langle 5\rangle} \mathbf{0}\\ &\xrightarrow{\surd} \epsilon\end{aligned}$$

Note that the termination timings of the two processes are different.

4 Algebraic Semantics for Optimizations

This section formulates an algebraic order relation which is suitable for verifying the correctness and effectiveness of optimized programs. It is an extension of the concept of temporal bisimulation [20] with the ability to compare two processes with respect to their relative execution speeds.

Definition 4.1 A binary relation $\mathcal{R} \subseteq (\mathcal{P}\times\mathcal{P})\times\mathcal{T}$ is a t-*optimization* prebisimulation over $\mathcal{P}$ if $(P_1, P_2) \in \mathcal{R}_t$ where $t \geq 0$. $(P_1, P_2) \in \mathcal{R}_{t'}$ is defined for all $\mu \in Act \cup \{\surd\}$ and $d \in \mathcal{T}$,

(i) $\forall P_1{}'$: $P_1 \overset{\langle d \rangle}{\Longrightarrow}\overset{\mu}{\Longrightarrow} P_1'$ *then* $\exists P_2{}'$: $P_2 \overset{\langle d+t' \rangle}{\Longrightarrow}\overset{\widehat{\mu}}{\Longrightarrow} P_2'$ *and* $(P_1', P_2') \in \mathcal{R}_0$

(ii) $\forall P_2{}'$, $P_2 \overset{\langle d \rangle}{\Longrightarrow}\overset{\mu}{\Longrightarrow} P_2'$ *then* $\exists d'$, $\exists P_1{}'$: $d' \geq d \dot{-} t'$ *and* $P_1 \overset{\langle d' \rangle}{\Longrightarrow}\overset{\widehat{\mu}}{\Longrightarrow} P_1'$ *and* $(P_1', P_2') \in \mathcal{R}_{t'-d+d'}$

where $P_1 \gtrsim_t P_2$ if there exist some t-optimization prebisimulations such that $(P_1, P_2) \in \mathcal{R}_t$. We call $\gtrsim_t$ t-*optimization* order. We shall often abbreviate $\gtrsim_0$ to $\gtrsim$. □

We briefly explain the above definition. $\mathcal{R}_t$ is a family of relations indexed by a non-negative time value t. Intuitively, t is the relative difference between the time of P_1 and that of P_2. $P_1 \gtrsim_t P_2$ starts with a prebisimulation indexed by t (i.e., $\mathcal{R}_t$) and can change t as the bisimulation proceeds only if $t \geq 0$. Therefore, $P_1 \gtrsim_t P_2$ means that P_2 precedes P_1 by t time units. We show several algebraic properties of the order relation below.

The informal meaning of $P_1 \gtrsim_0 P_2$ is that program P_1 is less improved than program P_2, in the sense that P_1 and P_2 have the same behaviors except for their internal computations, and P_2 can perform the behaviors *faster* than P_1. On the other hand, an optimized program is required to perform the same behavioral properties as the original program but faster. The relation enables us tostrictly prove whether an optimized program and its original one can perform the same behaviors and whether the optimized program can perform them faster than the original one.

Proposition 4.2 Let $P, P_1, P_2, P_3 \in \mathcal{P}$ and $t_1, t_2 \in \mathcal{T}$. Then

(1) $P \gtrsim P$

(2) If $P_1 \gtrsim_{t_1} P_2$ and $P_2 \gtrsim_{t_2} P_3$ then $P_1 \gtrsim_{t_1+t_2} P_3$. □

We can directly obtain that $P \gtrsim P$, and if $P_1 \gtrsim P_2$ and $P_2 \gtrsim P_3$ then $P_1 \gtrsim P_3$. Hence, $\gtrsim$ is a preorder relation.

Example 4.3 We show some basic examples of $\gtrsim$ as follows:

(1) Two sequential processes:

$$\langle 1 \rangle; \overline{a} \quad \gtrsim \quad \overline{a}$$

(2) Two parallel processes:

$$\langle 1 \rangle; \overline{a}; P_1 \parallel \langle 1 \rangle; \overline{b}; P_2 \quad \gtrsim \quad \overline{a}; P_1 \parallel \langle 1 \rangle; \overline{b}; P_2$$

(3) The following result says that the relation can compare two programs in synchronous communication setting. The program on the right is faster in a synchronously communicating with any environment.

$$\langle 3 \rangle; \overline{a}; \langle 2 \rangle; \overline{b} \quad \gtrsim \quad \langle 2 \rangle; \overline{a}; \langle 1 \rangle; \overline{b}$$
$$c.f. \quad \langle 3 \rangle; \overline{a}; \langle 2 \rangle; \overline{b} \quad \not\gtrsim \quad \langle 1 \rangle; \overline{a}; \langle 3 \rangle; \overline{b}$$

In a counterexample the right program is not faster when it communicates with program $\langle 4 \rangle; a; \langle 2 \rangle; b$.

(4) The left expression in the following inequality is given in Example 3.9.

$$(\langle 1 \rangle; \overline{a}; \langle 3 \rangle; b) \parallel (a; ((\langle 2 \rangle; \overline{b}; \langle 1 \rangle) \otimes \langle 4 \rangle))$$
$$\gtrsim \quad (\langle 1 \rangle; \overline{a}; \langle 3 \rangle; b) \parallel (a; \langle 4 \rangle; \langle 2 \rangle; \overline{b}; \langle 1 \rangle)$$

This says that the left and the right programs have the same behaviors and the right one can perform them faster than the left one. Therefore, the left program is an optimized program of the right one.

(5) Suppose that $a; ((\langle 2 \rangle; \overline{b}; \langle 1 \rangle) \otimes \langle 4 \rangle)$ interacts with $\langle 3 \rangle; \overline{a}; \langle 1 \rangle; b$. We have another inequality as follows:

$$(\langle 3 \rangle; \overline{a}; \langle 1 \rangle; b) \parallel (a; ((\langle 2 \rangle; \overline{b}; \langle 1 \rangle) \otimes \langle 4 \rangle))$$
$$\gtrsim \quad (\langle 3 \rangle; \overline{a}; \langle 1 \rangle; b) \parallel (a; \langle 2 \rangle; \overline{b}; \langle 1 \rangle; \langle 4 \rangle)$$

$$c.f. \quad (\langle 3 \rangle; \overline{a}; \langle 1 \rangle; b) \parallel (a; ((\langle 2 \rangle; \overline{b}; \langle 1 \rangle) \otimes \langle 4 \rangle))$$
$$\not\gtrsim \quad (\langle 3 \rangle; \overline{a}; \langle 1 \rangle; b) \parallel (a; \langle 4 \rangle; \langle 2 \rangle; \overline{b}; \langle 1 \rangle)$$

The following proposition means that $\gtrsim$ is preserved by all the constructors of the calculus.

Theorem 4.4 Let $P, P_1, P_2 \in \mathcal{P}$ and $t_1, t_2 \in \mathcal{T}$. Then

(1) If $P_1 \gtrsim P_2$ then $P_1; P \gtrsim P_2; P$, $P; P_1 \gtrsim P; P_2$, and $P \parallel P_1 \gtrsim P \parallel P_2$

(2) If $P_1 \gtrsim P$ and $P_2 \gtrsim P$ then $P_1 \oplus P_2 \gtrsim P$ and $P_1 \otimes P_2 \gtrsim P; P$

(3) If $P \gtrsim P_1$ and $P \gtrsim P_1$ then $P \gtrsim P_1 \oplus P_2$ and $P; P \gtrsim P_1 \otimes P_2$

(4) If $t_1 \geq t_2$ then $\langle t_1 \rangle \gtrsim \langle t_2 \rangle$ □

The above properties are important in constructing optimizing compilers, because they assert that the substitution of an optimized program for the original program can still be an improvement in any context. For example, a program can always be replaced by an optimized one of it in any context without changing any functional behaviors of the original program and the optimized one can perform the behaviors as fast as or faster than the original one. Therefore, the correctness and effectiveness of a compound optimization, which consists of more than one code optimization written in our language, can be guaranteed.

To prove the properties, we need to introduce the concept of confluent processes as studied by Milner and Tofts [11, 26].[2] Indeed all the processes in P are confluent because when they have any two possible actions, the occurrence of one will never preclude the other. That is, if a process has several possible actions, the processes derived by any sequences of the actions or the passage of time are properly related through $\gtrsim$.

On the other hand, this means that the calculus presented in this paper may not be able to express peculiar contexts that restrict the capability to execute a particular computation due to the passage of time or the other computation, for example *timeout* handling. This problem is originated from the semantics of the contexts instead of the calculus and the order relation.

5. Optimizing Transformation

The optimization order relation presented in the previous section provides a suitable method for verifying the correctness and effectiveness of optimized programs. This section constructs a general method for deriving optimized programs written in parallel programming languages. An optimization can be regarded as a program transformation. It transforms program code into better code which has the same behaviors as the original code and can perform them faster than the original one. This section tries to formalize various code optimizations as a rewriting system which transforms a program into an optimized one of the program. The system is defined in the form of inference rules corresponding to code optimizations for programs presented as expressions of the calculus or the specification language. The purpose of this section is to introduce basic optimizing transformations for programs of parallel object oriented languages. To save space, we leave the full theoretical investigation of the transformations to our other paper [25].

We first define basic optimizing transformation rules for syntactic constructions in the calculus. In the following rules, $P_1 \geqq P_2$ corresponds to the transformation of optimizing P_1 into P_2.

Definition 5.1 $\leqq$-theory is given by the following inference rules:

$$(1)\ \frac{P_1 \equiv P_2}{P_1 \geqq P_2} \qquad (2)\ \frac{P_1 = P_2}{P_1 \geqq P_2}$$

$$(3)\ \frac{P_1 \geqq P_2 \qquad P_2 \geqq P_3}{P_1 \geqq P_3} \qquad (4)\ \frac{t_1 \geq t_2}{\langle t_1 \rangle \geqq \langle t_2 \rangle}$$

$$(5)\ \frac{P_1 \geqq P_2}{P_1; P \geqq P_2; P} \qquad (6)\ \frac{P_1 \geqq P_2}{P; P_1 \geqq P; P_2}$$

$$(7)\ \frac{P_1 \geqq P_2}{P_1 \parallel P \geqq P_2 \parallel P} \qquad (8)\ \frac{P_1 \geqq P \qquad P_2 \geqq P}{P_1 \oplus P_2 \geqq P}$$

$$(9)\ \frac{P \geqq P_1 \qquad P \geqq P_1}{P \geqq P_1 \oplus P_2} \qquad (10)\ \frac{P_1 \geqq P \qquad P_2 \geqq P}{P_1 \otimes P_2 \geqq P; P}$$

$$(11)\ \frac{P \geqq P_1 \qquad P \geqq P_1}{P; P \geqq P_1 \otimes P_2}$$

where $\geqq \subseteq \mathcal{P} \times \mathcal{P}$. $P_1 \equiv P_2$ is given in Definition 3.4. $P_1 = P_2$ is an equation given as an additional axioms and is reflexive, transitive, and symmetric. □

The inequality above the horizontal line of each rule specifies the condition under which an optimization can be applied to a program, and the inequality below the line specifies the effect of applying the optimization to the program. For example, rule (5) means that if P_2 is an optimized program of P_1, then the sequential composition $P_2; P$ is still an optimized program of $P_1; P$.

It is worth mentioning the openness of the framework. A new optimization technique can easily be implemented by introducing the transformation rule corresponding to the technique. Hereafter we establish some derivations of $\leqq$-theory in order to treat specific optimization techniques. We define notations for such derivations. Let $\mathcal{F}, \mathcal{F}', \ldots$ range over $\leqq$-theory with some additional axioms and rules. We denote $\vdash_{\mathcal{F}} P_1 \geqq P_2$ if $P_1 \geqq P_2$ is provable in $\mathcal{F}$. We often denote $P_1 \geqq P_2$ if the concerned theory is obvious from the context. Given a set of axioms and rules, written as $\mathcal{A}_i$, a family of $\leqq$-theories, written as $\mathcal{F} + \mathcal{A}_i$, is the result of adding inequations as axioms to $\mathcal{F}$. We denote $\leqq$-theory with no additional axioms or rules as $\mathcal{F}_0$.

Definition 5.2 $\mathcal{A}_0$ is given as the following axioms and rules:

[2] We leave the full theoretical investigation of the order relation, including the proof, to our other paper [25].

$$\begin{array}{lrcl}
(1) & \langle t\rangle;(P_1 \parallel P_2) & = & \langle t\rangle;P_1 \parallel \langle t\rangle;P_2 \\
(2) & \langle t\rangle;(P_1 \oplus P_2) & = & \langle t\rangle;P_1 \oplus \langle t\rangle;P_2 \\
(3) & \langle t_1+t_2\rangle & = & \langle t_1\rangle;\langle t_2\rangle \\
(4) & (P_1 \oplus P_2);P_3 & = & (P_1;P_3) \oplus (P_2;P_3) \\
(5) & P_1;(P_2 \oplus P_3) & = & (P_1;P_2) \oplus (P_1;P_3) \\
(6) & (P_1 \oplus P_2) \parallel P_3 & = & (P_1 \parallel P_3) \oplus (P_2 \parallel P_3)
\end{array}$$

where $t, t_1, t_2 \in \mathcal{T}$ such that $t > 0$. □

$\mathcal{F}_0 + \mathcal{A}_0$-theory provides a general rewriting system which can optimize programs of many parallel languages. This is because $\mathcal{F}_0 + \mathcal{A}_0$ consists only of optimizing transformation rules which are satisfied independently of the fashion of interactions among parallel programs, for example synchronous communication, asynchronous communication, and shared variables.

The remainder of this section addresses several peculiar optimizations for parallel programs. They are formulated as supplementary transformation rules of $\geqq$-theory. For example, when interaction among parallel programs is assumed to be via one-way synchronous communication, we have the following equations:

Definition 5.3 $\mathcal{A}_1$ is given as the following axioms and rules:

$$\begin{array}{rl}
& \langle d\rangle;(\ell_1|\cdots|\ell_i|\cdots|\ell_m);P_1 \parallel (\ell'_1|\cdots|\ell'_j|\cdots|\ell'_n);P_2 \\
= & \langle d\rangle;((\ell_1|\cdots|\ell_{i-1}|\ell_{i+1}|\cdots|\ell_m);P_1 \\
& \quad \parallel (\ell'_1|\cdots|\ell'_{j-1}|\ell'_{j+1}|\cdots|\ell'_n);P_2)
\end{array}$$

where $\ell_i = \overline{\ell'_j}$. □

From $\mathcal{A}_1$, we directly obtain $\langle d\rangle;\ell;P_1 \parallel \overline{\ell};P_2 = \langle d\rangle;(P_1 \parallel P_2)$.

Definition 5.4 $\mathcal{F}_1$-theory is given by adding the following rules to $\mathcal{F}_0$.

(1)

$$\frac{(P_1 \parallel P_4);(P_2;P_3 \parallel P_5) \geqq (P_3;P_1 \parallel P_4);(P_2 \parallel P_5)}{((P_1;\overline{\ell};P_2) \otimes P_3) \parallel P_4;\ell;P_5 \geqq P_3;P_1;\overline{\ell};P_2 \parallel P_4;\ell;P_5}$$

(2)

$$\frac{(P_3;P_1 \parallel P_4);(P_2 \parallel P_5) \geqq (P_1 \parallel P_4);(P_2;P_3 \parallel P_5)}{((P_1;\overline{\ell};P_2) \otimes P_3) \parallel P_4;\ell;P_5 \geqq P_1;\overline{\ell};P_2;P_3 \parallel P_4;\ell;P_5}$$

□

The two rules decide how computative programs are optimized. The left program, $((P_1;\overline{\ell};P_2) \otimes P_3)$, has computative subprograms. This means that $P_1;\overline{\ell};P_2$ and P_3 are commutable programs in a parallel program. After performing program P_1, the former performs action $\overline{\ell}$ and then behaves as P_2 and the latter behaves as P_3. On the other hand, the right program, $P_4;\ell;P_5$, may still execute program P_4 before performing ℓ. The first rule means that when $(P_3;P_1 \parallel P_4);(P_2 \parallel P_5)$ is faster than $(P_1 \parallel P_4);(P_2;P_3 \parallel P_5)$, $(P_1;\overline{\ell};P_2) \otimes P_3$ should be optimized as program $P_3;P_1;\overline{\ell};P_2$. The second rule means that when $(P_1 \parallel P_4);(P_2;P_3 \parallel P_5)$ is faster than $(P_3;P_1 \parallel P_4);(P_2 \parallel P_5)$, $(P_1;\overline{\ell};P_2) \otimes P_3$ should be optimized as program $P_1;\overline{\ell};P_2;P_3$. These rules are practical for optimizing programs of parallel languages because they can drastically reduce the temporal cost of synchronizations among parallel programs.

One of the most effective optimizations for parallel computation is to parallelize sequential programs. Although this framework does not intend to extract independent pieces which can be performed concurrently, form a program, it can formalize parallelizing optimizations as transformation rules as follows:

Definition 5.5 $\mathcal{F}_2$-theory is given by adding the following rules to $\mathcal{F}_1$.

(1)

$$\frac{}{(P_1 \parallel P_2) \otimes (P_3 \parallel P_4) \geqq (P_1;P_3) \parallel (P_2;P_4)}$$

where $\mathcal{L}(P_1), \mathcal{L}(P_2), \mathcal{L}(P_3), \mathcal{L}(P_4) = \emptyset$.

(2)

$$\frac{}{P_1 \otimes P_2 \geqq P_1 \parallel P_2}$$

where $\mathcal{L}(P_1), \mathcal{L}(P_2) = \emptyset$. □

The above rules seem to be restricted because they are designed for rules which are valid at various interprocess communication settings. However, we can get some more practical rules to parallelize sequential processes when communications among parallel processes can be restricted to within peculiar fashions, for example remote procedure call and object oriented computation.

We show that the transformation rules presented in this section are sound on the optimization order relation, $\gtrsim$.

Theorem 5.6 Let $\mathcal{F}$ be $\mathcal{F}_0$, $\mathcal{F}_0 + \mathcal{A}_0$, $\mathcal{F}_0 + \mathcal{A}_0 + \mathcal{A}_1$, $\mathcal{F}_1 + \mathcal{A}_0 + \mathcal{A}_1$, and $\mathcal{F}_2 + \mathcal{A}_0 + \mathcal{A}_1$.

$$P_1 \gtrsim P_2 \quad \text{if} \quad \vdash_{\mathcal{F}} P_1 \geqq P_2$$

□

This result is important because the rules can be used as correctness- and effectiveness-preserving transformations, in the sense of the optimization order relation studied in the previous section. Consequently, we can ensure that all the programs derived from the rules of $\geqq$-theory are more optimized than their original ones.

Unfortunately, we cannot obtain the completeness of $\leqq$-theory.

Example 5.7 We show how to optimize parallel programs by using these optimizing transformations. Consider two parallel programs. They are specified as follows:

$$((\langle 1\rangle; \overline{a}; \langle 2\rangle; b) \otimes \langle 4\rangle) \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c}$$

The left program consists of two commutative subprograms: $\langle 1\rangle; \overline{a}; \langle 2\rangle; b$ and $\langle 4\rangle$. The former is a program which sends action a after one time unit. Next, it performs an internal execution for two time units and then receives action b. The latter is a program which performs an internal execution for four time units. The right program performs an internal execution for three time units and then receives action a. After five time units, it sends action b and then behaves as $\langle 6\rangle; \overline{c}$. We derive an optimized program of the above parallel programs by using the optimizing transformation rules as follows:

$$\begin{aligned} &\langle 1\rangle; \overline{a}; \langle 2\rangle; b; \langle 4\rangle \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c} \\ &\quad \geqq \quad \langle 4\rangle; \langle 1\rangle; \overline{a}; \langle 2\rangle; b \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c} \end{aligned}$$

because we apply $\mathcal{A}_0$ and $\mathcal{A}_1$ to the left program:

$$\begin{aligned} &\langle 1\rangle; \overline{a}; \langle 2\rangle; b; \langle 4\rangle \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c} \\ &\quad = \quad \langle 1\rangle; \tau; (\langle 2\rangle; b; \langle 4\rangle \parallel \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c}) \\ &\quad = \quad \langle 1\rangle; \tau; \langle 5\rangle; (\langle 4\rangle \parallel \langle 6\rangle; \overline{c}) \\ &\quad = \quad \langle 1\rangle; \tau; \langle 5\rangle; \langle 6\rangle; \overline{c} \\ &\quad = \quad \cdots \end{aligned}$$

Moreover, we apply $\mathcal{A}_0$ and $\mathcal{A}_1$ to the right program in the same way:

$$\begin{aligned} &\langle 4\rangle; \langle 1\rangle; \overline{a}; \langle 2\rangle; b \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c} \\ &\quad = \quad \langle 4\rangle; \langle 1\rangle; (\langle 2\rangle; b \parallel \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c}) \\ &\quad = \quad \cdots \end{aligned}$$

We easily prove the above inequality using $\mathcal{F}_0$. Therefore, from $\mathcal{F}_0$ and $\mathcal{F}_1$ we can derive the following inequation:

$$\begin{aligned} &((\langle 1\rangle; \overline{a}; \langle 2\rangle; b) \otimes \langle 4\rangle) \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c} \\ &\quad \geqq \quad \langle 4\rangle; \langle 1\rangle; \overline{a}; \langle 2\rangle; b \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c} \end{aligned}$$

The right program is an optimized of program $((\langle 1\rangle; \overline{a}; \langle 2\rangle; b) \otimes \langle 4\rangle) \parallel \langle 3\rangle; a; \langle 5\rangle; \overline{b}; \langle 6\rangle; \overline{c}$. It is worth mentioning that a series of applying the transformation rules can be regarded as an automatic derivation of optimized programs in optimizing compilers.

Remark The reader may wonder why we restrict our attention to a few conservative optimization techniques in this paper. The reason is that the framework aims to ensure the correctness and temporal effectiveness of all the programs derived from the rules. However, the framework has the potential to cope with various optimization techniques by adding transformation rules corresponding to the techniques. To optimize more comprehensive expressions, it is necessary to construct an *aggressive* framework which can analyze runtime properties of programs but is not always sound.

6. Related Work

We briefly survey related work. A number of code optimization techniques have been studied and used for parallel programming language compilers. However, the specification and verification of optimizations have not been well advanced. Serval researchers have explored formal frameworks for verifying the correctness and effectiveness of compiler optimization techniques, for example [4, 6, 8, 10, 18]. However, most of them have been essentially designed for sequential programming languages. A few frameworks have the ability to reason about optimizations techniques for parallel languages. Nielsen in [16] studied a control flow analysis for Concurrent ML. It can predict the communication topology of programs presented as terms of a process calculus and a type system. Wand in [27] explored the semantics of a parallel functional language which is applicable to the verification of compiler correctness. The framework presented in this paper is unique among existing related works in having the ability to quantitatively analyze temporal effectiveness of optimizations for parallel programs. The author in [21] proposed a framework to define semantics for parallel programming languages. The framework is characterized by specifying temporal semantics of programs including execution time but can not verify the temporal effectiveness of code optimizations for parallel programming languages. Also, the author in [24] proposed a process calculus to specify and verify optimized programs written by using object oriented languages. Unfortunately, it is essentially dependent on some peculiar features of object orientation and thus is not available in formalizing arbitrary parallel programming languages, unlike the framework presented in this paper.

In the literature of process calculi. There have indeed been many time extended process calculi, for example [3, 20, 28]. Most of the calculi have been equipped with time-sensitive equivalence relations which equate two processes only when they are behaviorally and temporally equivalent. But only a few of them provide a method to compare temporal costs of parallel systems, for example [7, 15, 23]. Moller and Tofts in [15] and Jenner and Vogler in [7] proposed preorder relations over parallel processes with respect to their relative speeds. However, unlike ours, the semantics of the relations assumes to permit an executable communication to be suspended for arbitrary periods of time, and thus cannot exactly predict the synchronization time of parallel programs. In [23] the author studied a speed sensitive order relation but the relation is designed for verifying asynchronously communicating processes. Arun-Kumar and Hennessy [1], Chen and Gorrieri in [2], and Natarajan and Cleaveland in [14] proposed approaches to relate processes according to the number of necessary internal actions. However, they do not take the cost of synchronization into consideration and thus are not available in analyzing optimizations for parallel programs.

7. Conclusion

This paper proposed a theoretical framework for verifying and deriving optimized parallel programs. The framework consists of a specification language, an algebraic order relation, and optimizing transformation rules. The language is formulated as an extended process calculus. The relation can state as an optimized program has the same behaviors as the original one, and the optimized program can perform the behaviors faster than the original one. We also formalized several code optimizations as transformation rules between a program and its optimized one. The advantage of the transformation rules is that the correctness and effectiveness of optimized programs derived from the rules can always be ensured.

There are many issues in this paper that we leave for future work. One of the most important issues is to apply this framework to various code optimizations used in real compilers for parallel programming languages. Also, the framework intentionally makes minimal assumptions about the runtime environment of programs. However, there are many optimization techniques that closely depend on computer architecture, task scheduling policy, and machine instruction. We plan to study the formalization of machine-specific and resource-aware optimizations in future. The calculus presented in this paper consists of only the minimal constructors. The author in [21] developed a technique for defining the functional and temporal semantics of enriched parallel object-oriented languages. We plan to introduce that technique into this framework in order to analyze programs written conventional programming languages, for example FORTRAN and C. Moreover, type systems often enable a program to be optimized well. We are interested in introducing type-based approaches to this framework. In particular, some researchers have recently studied type theories for behaviors of processes, for example [9, 17, 19]. Their approaches allow this framework to formalize more optimization techniques. A series of applications of the transformation rules can be regarded as an automatic derivation of optimized programs in optimizing compilers. We believe that this framework can give theoretical and practical benefits to the design and implementation of real optimizing compilers for parallel programming languages.

Acknowledgements

We would like to thank anonymous referees for providing many constructive and valuable suggestions.

References

[1] Arun-Kumar, S. and Hennessy, M., *An Efficiency Preorder for Processes*, Acta Informatica, Vol.29, pp.737-760, 1992.

[2] Chen. X.J., Corradini, F., and Gorrieri, R., *A Study on the Specification and Verification of Performance Properties*, Proceedings of Algebraic Methodology and Software Technology, LNCS, Vol.1011, Springer-Verlag, 1996.

[3] Davies, J. W., *Specification and Proof in Real-Time CSP*, Cambridge University Press, 1994.

[4] He, J. and Bowen, J., *Specification, Verification and Prototyping of an Optimized Compiler*, Formal Aspects of Computing, Vol.6, pp.643-658, 1994.

[5] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice Hall, 1985.

[6] Hoare, C.A.R., Jifeng, H., and Sampaio, A., *Normal Form Approach to Compiler Design*, Acta Informatica, Vol.30, pp.701-739, 1993.

[7] Jenner, L. and Volger, W., *Faster Asynchronous Systems in Dense Time*, Proceedings of ICALP'96, LNCS, Vol.1099, pp.75-86, Springer-Verlag, 1996.

[8] Jifeng, H. and Bowen, J., *Specification, Verification and Protyping of an Optimized Compiler*, Formal Aspects of Computing, Vol.6, No.6, pp.643-658, 1994.

[9] Kobayashi, N., Pierce, B.C., and Tuner, D.N., *Linearity and the π-calculus*, Proceedings of POPL'96, 1996.

[10] Levin, V, *Algebraically Provable Specification of Optimized Compilations*, Proceedings of Formal Methods in Pargramming and Their Applications, LNCS, Vol. 735, 1993.

[11] Milner, R., *Communication and Concurrency*, Prentice Hall, 1989.

[12] Milner, R., *Functions as Processes*, Mathematical Structure in Computer Science, Vol.2, No.2, pp.119-141, 1992.

[13] Milner, R., Parrow. J., and Walker, D., *A Calculus of Mobile Processes*, Information and Computation, Vol.100, pp.1-77, 1992.

[14] Natarajan, V., and Cleaveland, R., *An Algebraic Theory of Process Efficiency*, Proceedings of LICS'96, pp.63-72, June, 1996.

[15] Moller, F. and Tofts, C., *Relating Processes with Respect to Speed*, Proceedings of CONCUR'91, LNCS, Vol.527, Springer-Verlag, 1991.

[16] Nielsen, H.R. and Nielsen, F., *Communication Analysis for Concurrent ML*, in ML with Concurrency, pp.186-235. Springer-Verlag, 1996.

[17] Nierstrasz, O. M., *Regular Types for Active Objects*, Proceedings of OOPSLA'93, October, p1-15, 1993.

[18] Sampio, A., *An Algebraic Approach to Compiler Design*, World Scientific, 1997.

[19] Sangiorgi, D., *The Name Discipline of Receptiveness*, Technical Report INRIA-Sophia Antipolis, 1997.

[20] Satoh, I., and Tokoro, M., *A Formalism for Real-Time Concurrent Object-Oriented Computing*, Proceedings of OOPSLA'92, p315-326, October, 1992.

[21] Satoh, I., and Tokoro, M., *Semantics for a Real-Time Object-Oriented Programming Language*, Proceedings of IEEE International Conference on Computer Languages, p159-170, May, 1994.

[22] Satoh, I. and Tokoro, M., *A Formalism for Remotely Interacting Processes*, Proceedings of TPPP'94, LNCS, Vol.907, pp.216-228, Springer-Verlag, 1995.

[23] Satoh, I., and Tokoro, M., *Time and Asynchrony in Interactions among Distributed Real-Time Objects*, Proceedings of ECOOP'95, LNCS 952, pp.331-350, Springer-Verlag, 1995.

[24] Satoh, I., *Speed-Sensitive Orders for Communicating Processes*, Concurrency Theory and Application'96, pp.23-39, A revised version in Research Institute for Mathematical Sciences Report 996, Kyoto University, 1997.

[25] Satoh, I. *An Algebraic Framework for Optimizing Parallel Programs*, a typescript, to appear as a technical report, Ochanomizu University, 1998.

[26] Tofts, C., *Proof Methods and Pragrmatics for Parallel Programming*, PhD thesis, University of Edinburgh, 1990.

[27] Wand, M., *Compiler Correctness for Parallel Languages*, Proceedings of Symposium on Functional Languages and Computer Architecture, 1995.

[28] Wang, Y., *CCS + Time = an Interleaving Model for Real Time Systems*, Proceedings of ICALP'91, LNCS, Vol.510, Springer-Verlag, 1991.