Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 49–63
www.elsevier.com/locate/entcs

Generating Java Compiler Optimizers Using Bidirectional CTL

Ling Fang [1,2] Masataka Sassa[3]

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1, O-okayama, Meguro-ku, Tokyo 152-8552, Japan

Abstract

There have been several research works that analyze and optimize programs using temporal logic. However, no evaluation of optimization time or execution time of these implementations has been done for any real programming language. In this paper, we present a system that generates a Java optimizer from specifications in temporal logic. The specification is simpler, and the generated optimizers run more efficiently than previously reported work. We implemented a new model checker for a bidirectional CTL (computational tree logic) called CTL$_{bd}$, which is equivalent to CTL-FV [9] after removing free variables. The model checker can check future and past temporal CTL operators symmetrically without any conversion. We also present a new specification language based on the bidirectional CTL that can express typical optimization rules very naturally. By adding rewriting conditions to allow for temporary variables and considering real-world language features such as exceptions, the system can perform optimization of Java programs. So far, a compiler optimizer using temporal logic was assumed to be impractical, because it consumes too much time. However, with our method, the generated Java compiler optimizer can compile seven of the SPECjvm98 benchmarks with a compile time from 4 seconds to 4 minutes.

Keywords: compiler optimization, temporal logic, CTL

1 INTRODUCTION

In compiler design, code optimization is one of the most important passes, improving execution speed and spatial efficiency of the target code [1].

Current optimizers are almost always implemented by some kind of programming language. However, the approach of implementing optimizers by CTL (computational tree logic) [6], a branching temporal logic, has attracted interest in recent years. This approach has two main advantages.

- The transformations are easier to write and prototype. They can be achieved by writing several lines of specification language rather than many hundreds of lines

[1] Earlier version of parts of this article appeared at a conference of JSSST in Japanese.

[2] Email: fang3@is.titech.ac.jp Fax: +81-3-5734-3210

[3] Email: sassa@is.titech.ac.jp Fax: +81-3-5734-3210

doi:10.1016/j.entcs.2007.09.007

of code.

- The transformations can be formally analyzed and proved because they are more simply expressed.

Compiler optimization by CTL can concisely express a lot of classic optimization by using the following specification language (conditional rewrite rule), which is denoted as follows.

$$I \Longrightarrow I' \text{ if } \phi$$

In our work, we actually use a kind of bidirectional CTL which we call CTL$_{bd}$. CTL$_{bd}$ is based on CTL-FV [7], in which past temporal operators can be used symmetrically with future temporal operators and free variables when these are introduced. CTL$_{bd}$ is equivalent to CTL-FV after removing free variables.

We implemented a model checker that directly handles CTL$_{bd}$. Past temporal operators are checked symmetrically with future temporal operators. Because the process time used to convert to μ-calculation or from PCTL to NCTL [10] is not incurred and the formulas never become more complex because of the elimination of past temporal operators, our model checker is very efficient.

Before model checking, *free variables* [4] must be bound. Therefore when there are a lot of free variables in the conditional expression, processing time becomes unrealistic. We clarify through experiment the fact that using the node numbers of the Kripke structure as free variables, as done in all previous work [9] [2] [19], will greatly increase the optimization time. Thus, formalization of optimization should be done using the least number of free variables.

The specification language we have developed does not refer to the node number of the Kripke structure. What the model checker calculates is not the instruction of a specific number but sets of instructions that satisfy the same condition. Therefore, it becomes very easy to describe a complex rewrite rule that rewrites many instructions. Moreover, efficiency is improved as the free variables corresponding to the node number of the Kripke structure are omitted.

So far, optimizers with temporal logic have been assumed to be impractical because of the amount of processing time needed. By adding some processing for real-world language features, we obtained several typical optimization phases for a Java language compiler, the performance of which is now close to optimizers that use traditional algorithms. In our research, seven of the SPECjvm98 benchmarks were able to be optimized in a time ranging from 4 seconds to 4 minutes using the aforementioned improvement.

Additionally, our specification allows simultaneous transformations of several points in the program using temporary variables, and it can be used in a real world compiler. Thus, our optimizer can perform several optimizations that were not performed in previous work using CTL [9] [2] [19].

We think that our implementation is the first realistic Java compiler optimizer with temporal logic. Moreover, insights into existing problems, and techniques for shortening the optimization time and the recommended style of its specification

[4] In this paper, free variables denote the free variables of logical formulas. See section 6. They should not be confused with variables in programs.

were acquired.

2 CTL$_{bd}$

CTL$_{bd}$ is a temporal logic in which past temporal operators are introduced symmetrically with future temporal operators.

CTL$_{bd}$ has past temporal operators $\overleftarrow{A}$ and $\overleftarrow{E}$ as well as the usual quantifiers A and E, made by reversing A and E.

A CTL$_{bd}$ formula is either a *state formula* ϕ or a *path formula* ψ, generated by the following grammar with nonterminals ϕ and ψ, terminals $true$, $false$ and $pred(x_1, \ldots, x_n) \in Pred$, start symbol ϕ, and the following productions.

Syntax

The syntax of CTL$_{bd}$ is:

$\phi ::= true \mid false \mid pred(x_1, \ldots, x_n)$
$\quad \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi$
$\quad \mid E\psi \mid A\psi \mid \overleftarrow{E}\psi \mid \overleftarrow{A}\psi$
$\psi ::= X\phi \mid \phi \; U \; \phi$

The standard abbreviations, e.g., $F\phi \equiv true \; U \; \phi$, $G\phi \equiv \neg F \neg\phi$ and $\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 U \neg\phi_2)$ hold as well.

Semantics

The semantics of CTL$_{bd}$ is given on the Kripke structure.

A Kripke structure K is a triple (S, R, L). S is a set of states, $R \subseteq S \times S$ is the transition relation, and $L : S \to 2^{Pred}$ is a function that maps each state to a set of predicates that are true for that state.

A path from s_0 in K is the infinite sequence of states $\pi = (s_0, s_1, \ldots)$ such that $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. A backward path from s_0 is a sequence such that $\forall i \geq 0 : (s_{i+1}, s_i) \in R$.

$K, s \models \phi$ denotes that the value of logical formula ϕ is true in state s in a Kripke structure K. K can be omitted if it is obvious. Relation $\models$ is defined as follows.

State Formulae:

$s \models true$ iff $true$
$s \models false$ iff $false$
$s \models pred(x_1, x_2, \ldots, x_n)$ iff $pred(x_1, x_2, \ldots, x_n) \in L(s)$
$s \models \neg\phi$ iff not $s \models \phi$
$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$
$s \models \phi_1 \vee \phi_2$ iff $s \models \phi_1$ or $s \models \phi_2$

$s \models E\psi$ iff $\exists path(s = s_0 \to s_1 \to s_2 \ldots) : (s_i)_{i \geq 0} \models \psi$
$s \models A\psi$ iff $\forall path(s = s_0 \to s_1 \to s_2 \ldots) : (s_i)_{i \geq 0} \models \psi$
$s \models \overleftarrow{E}\psi$ iff $\exists path(\ldots s_2 \to s_1 \to s_0 = s \ldots) : (s_i)_{i \geq 0} \models \psi$
$s \models \overleftarrow{A}\psi$ iff $\forall path(\ldots s_2 \to s_1 \to s_0 = s \ldots) : (s_i)_{i \geq 0} \models \psi$

Path Formulae:

$(s_i)_{i\geq 0} \models X\phi$ iff $s_1 \models \phi$

$(s_i)_{i\geq 0} \models \phi_1 U \phi_2$ iff

$$\exists k \geq [s_k \models \phi_2 \wedge \forall i : [0 \leq i < k \implies s_i \models \phi_1]]$$

A CTL$_{bd}$ is associated with two tree structures. One is a *CTL tree* (we sometimes write CTL instead of CTL$_{bd}$ where it is not ambiguous) that is expanded from the forward transition relation of the Kripke structure, and the other one is a $\overleftarrow{CTL}$ *tree* that is expanded from the backward transition relation of the Kripke structure. An example is shown in Figure 1.

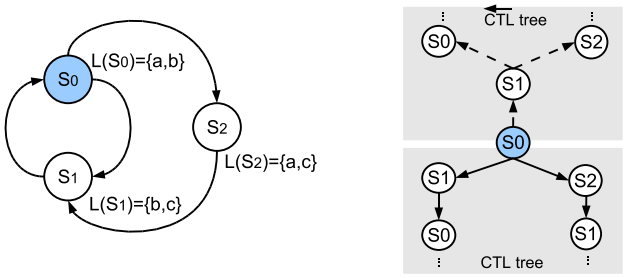

Fig. 1. Kripke structure (left) and its CTL (infinite) trees (right) for S_0

3 Control Flow Model

We explain the control flow model using a simple example. Here we assume a simple language with no procedures for explanatory purposes.

$\pi = read\ x; I_1; I_2; \ldots I_{m-1}; write\ y$

An example is shown on the left of Figure 2. The instructions are labeled by $n \in Node_\pi = \{0, 1, 2, \ldots, m\}$.

The control flow model of code π is defined similarly to that in compilers, as shown on the center of Figure 2. We can attach label $L_{\pi(n)}$ for each node, which represents the set of properties that hold at n.

The $L_{\pi(n)}$ corresponding to the code and control flow model is shown on the right of Figure 2.

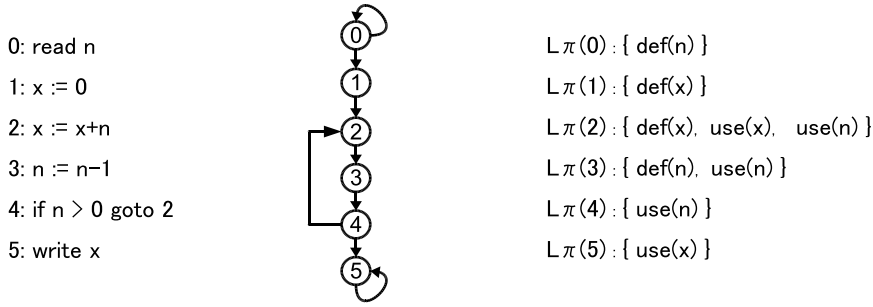

Fig. 2. Example of code and its control flow model

4 CTL$_{bd}$ Model Checker

This section describes the CTL$_{bd}$ model checker implemented in our work.

The algorithm of the CTL$_{bd}$ model checker is an extension of model checkers used in previous work [2]. Checking the future temporal logic operations is calculated from the CTL tree by the usual algorithm. However, checking past temporal logic operations is calculated from a $\overleftarrow{CTL}$ tree in a symmetric fashion to that of the future operators by simply reversing the direction.

It is an explicit state model checker. It is efficient because it can treat past temporal logic without any transformation.

We constructed a new model checker instead of modifying an existing model checker because we think our model checker can easily handle the program features. Also, creating our own model checker will make it easy to extend its functionality in the future such as improving the algorithm and data structures to make it more efficient, e.g., by using bit vectors (or partial evaluation).

Analysis of CTL$_{bd}$ Formula

A CTL$_{bd}$ formula can be expressed by a tree structure, which we call the *CTL$_{bd}$ syntax tree* (note that it is different from the CTL$_{bd}$ tree). The leaves of the tree are atomic predicates.

Figure 3 shows the CTL$_{bd}$ formula (top) and its syntax tree (bottom). Table 1 shows the partial formulas corresponding to each node of the CTL$_{bd}$ syntax tree.

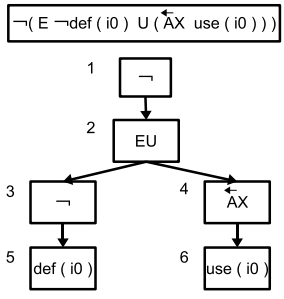

Fig. 3. CTL$_{bd}$ formula and CTL$_{bd}$ syntax tree

No.	Op	Child	Partial Formula
1	$\neg$	2	$\neg(E\ \neg def(i_0)\ U \overleftarrow{A} X(use(i_0)))$
2	EU	3 4	$E\ \neg def(i_0)\ U \overleftarrow{A} X(use(i_0))$
3	$\neg$	5	$\neg def(i_0)$
4	$\overleftarrow{A} X$	6	$\overleftarrow{A} X(use(i_0))$
5	ap	$def(i_0)$	$def(i_0)$
6	ap	$use(i_0)$	$use(i_0)$

Table 1. CTL$_{bd}$ syntax tree nodes and partial formulas of Figure 3

Model Checking of CTL$_{bd}$

Each partial formula ϕ_n should be calculated to know whether it is satisfied at each state s. Namely, if we denote the truth value of partial formula ϕ_n at state s_i by $label(\phi_n, s_i)$, we calculate the truth value of these labels as follows.

$$label(\phi_n, s_i) = true \text{ iff } s_i \models \phi_n$$

For that purpose, for each state s of the Kripke structure and for each ϕ_n, a bottom-up calculation is done from the leaves to the root of the CTL$_{bd}$ syntax tree.

Because the results of the model checking will be used in the following rewriting process, the results needed in the rewriting process are stored in a data structure

during the model checking. This data structure is a set corresponding to the nodes of the CTL_{bd} syntax tree.

Computational Complexity of Model Checking

There is an algorithm to determine whether a CTL formula ϕ is true in a state s of the structure $M = (S, R, L)$ that runs in time $O(|\phi|(|S| + |R|))$ [4]. If we denote the number of instructions by $n1$, and the number of nodes of a CTL_{bd} syntax tree by $n2$, then the computational complexity of model checking is approximately $O(n1 \times n2)$.

5 The Specification Language for Optimization

This section describes our language that can be used to specify optimizing transformations. The language is very simple, but it can express many standard compiler optimizations naturally.

5.1 Composition of the Specification for Optimization

The specification of optimization consists of three parts: **MATCH**, **CONDITION**, and **PROCESS**. The following, Table 2, shows the format of the optimization specification language (left) and an example specification for dead code elimination (right).

MATCH	**MATCH**
$\langle var \rangle := \langle expr \rangle$	$v := e$
CONDITION	**CONDITION**
$point\_\langle str \rangle : \langle CTL_{bp}\ formula \rangle$	$point\_delete :$
$edge\_\langle str \rangle : point\_\langle str \rangle \rightarrow point\_\langle str \rangle$	$\neg EX(E\neg def(v)Uuse(v))$
PROCESS	**PROCESS**
$point\_\langle str \rangle : \langle Comand \rangle \langle instruction \rangle$	$point\_delete :\ Delete\ v := e$
$point\_\langle str \rangle :\ Replace\ \langle expr \rangle \rightarrow \langle expr \rangle$	
$edge\_\langle str \rangle :\ EdgeSplit\ \langle instruction \rangle$	

Table 2
Specification format (left) and an example for dead code elimination (right)

The MATCH part specifies the pattern of instructions that are the target for optimization. It binds *free variables* written in the CONDITION part to variables or expressions in the program. In the above example, v and e are free variables denoting program variable and expression, respectively. Binding $\{v \mapsto x, e \mapsto x + n\}$ is done at the MATCH stage of the program as shown in Figure 2. This policy avoids useless bindings such as $\{v \mapsto n, e \mapsto x + n\}$, which is contained in the combination but has no corresponding instruction $n = x + n$ in the program.

In the CONDITION part, *conditional formulas* and *partial formulas* can be written. $point\_\langle string \rangle$ and $edge\_\langle string \rangle$ are called *(partial) formula names*. Conditional formulas are the conditions that must hold when an optimizing transformation is to be performed. Conditional formulas are named so that they can be referred to in the PROCESS part.

When a conditional formula is long and difficult to understand, it can be written by subdividing it into several partial formulas, which are also given names. Conditions about edges can also be written. They are called *edge conditions*.

The PROCESS part states how to process the instructions or edges that satisfy the conditional formulas in the CONDITION part. The names written before the ":" correspond to the names of the conditional formulas before the ":" in the CONDITION part. In the above example, "$v := e$" is deleted at the point where conditional formula *point_delete* holds. We can also introduce temporary variables where necessary.

The name of a formula (i.e. the name before ":") in the CONDITION or PROCESS part is different from the number associated with an instruction, i.e. the node number of the Kripke structure, in previous work [9] [2]. In our system, such a name is not a free variable or the node number of the Kripke structure and need not be bound before use. This name represents the set of instructions satisfying the same conditional formula. Similarly, what the model checker calculates is not a specific instruction but a set of instructions satisfying the conditional formula. As a result, it becomes very easy to describe the process of rewriting many instructions that satisfy several conditional formulas at the same time. Moreover, efficiency can be improved because free variables for instruction number are now omitted. This is one of the main differences from previous research. Detailed explanations can be found in [5].

5.2 Formalization of Optimization with Our Specification Language in CTL$_{bd}$

This section describes the formalization of compiler optimization using our specification language.

In our research, we can write a specification language in two ways. One is based on the meaning of optimization written in CTL$_{bd}$ similar to previous work [9], the other is based on the dataflow equations.

Specification Based on the Optimization Condition in CTL Formulas

This is the same as the specification in previous work [9] [2] [19] , but it is necessary to take features and efficiency into consideration when dealing with a real-world language such as temporary variable. The specification for dead code elimination mentioned above can be referred to as an example.

Specification Based on the Dataflow Equation

The crucial connection between model checking and dataflow analysis was made by Steffen [16].

Complex optimization like partial redundancy elimination is needed for real optimizing compilers. Many conditional formulas are necessary to specify it. The system must rewrite a set of instructions that satisfy the same conditional formula at the same time. Writing specifications in CTL$_{bd}$ from scratch considering the meaning and condition of optimization is difficult.

For such complex optimizations, specifications based on the dataflow equation are much easier than writing them from scratch. Partial redundancy elimination

has been investigated for many years, and many algorithms exist. We adopted the method of Paleri [13] and formalized it very easily. The CTL formula we write is almost a one-to-one mapping of the original dataflow equations with only a slight modification [5].

6 Free Variables

A *free variable* is a variable that appears in a conditional formula that is not yet bound to a specific variable or expression of program. The use of free variables was first introduced into CTL in Lacey's thesis [8].

Free variables must be bound to actual variables or expressions in the program before handling the program.

Because model checking is done on each of the combinations of bindings, the number of free variables greatly influences processing time, i.e. the optimization time of the system.

Let n : number of free variables in conditional formulas, m : number of objects that can be the target of binding of free variables in conditional formulas, such as variables or expressions in the program. Then, the *computational complexity* of free variable binding is $O(m^n)$.

Free variables has been adopted by most previous work [9] [2] [19], as well as ours, and it seems very convenient and expressible. However, as mentioned above, binding of free variables will cause an exponential computational complexity. An example of time explosion caused by free variables will be shown in section 8. There, we use different formulas for copy propagation, one containing 2 free variables that can be bound at the MATCH stage, and another containing 4 free variables with only 2 of them able to be bound at the MATCH stage. The resulting time explosion exhibited by the latter case will be shown later. It shows that the introduction of free variables needs to be avoided as much as possible in practical compiler optimizers. We focused on eliminating unnecessary free variables when we made our optimizers.

The question of how to make a compiler optimizer in CTL$_{bd}$ with fewer free variables is the subject of our planned future research.

7 Compiler Optimizer with CTL$_{bd}$

Figure 4 shows an outline of our optimization system. Our compiler optimizer with CTL$_{bd}$ is composed of three parts: the preprocessing part, the model checker, and the rewrite part. The preprocessing part binds free variables in the conditional formulas to the variables or expressions in the program after reading the source program and transforming it into a kind of 3-address intermediate code. The model checker calculates the points and edges of the program that satisfy the conditional formulas. The rewrite part applies optimizing rewrite rules to the result of the model checking part, and outputs the optimized program.

The system works within Soot [17], which is a Java optimization framework. We use Soot to get 3-address intermediate code Jimple and some utilities for analysis.

Figure 5 is an example of rewriting for partial redundancy elimination using our system. The program before optimization is shown on the left-hand side, and the program after optimization is shown on the right-hand side.

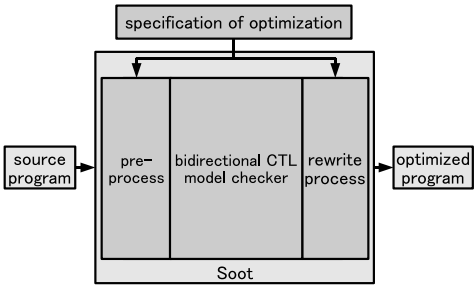

Fig. 4. Outline of the optimization system

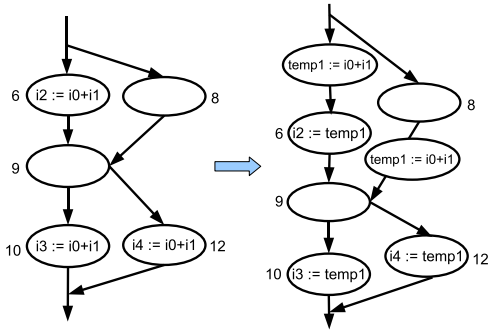

Fig. 5. Example of rewriting

Figure 6 is a simple example of model checking just for explanatory purposes, but is not part of a real optimizer. The source program (left) is optimized using the CTL$_{bd}$ syntax tree (right), which is made from the CTL$_{bd}$ formula (center). The nodes of the CTL$_{bd}$ syntax tree can hold names of formulas or partial formulas if necessary. The result of model checking will be put into sets corresponding to such *(partial) formula names* e.g. *point_a* in the right figure.

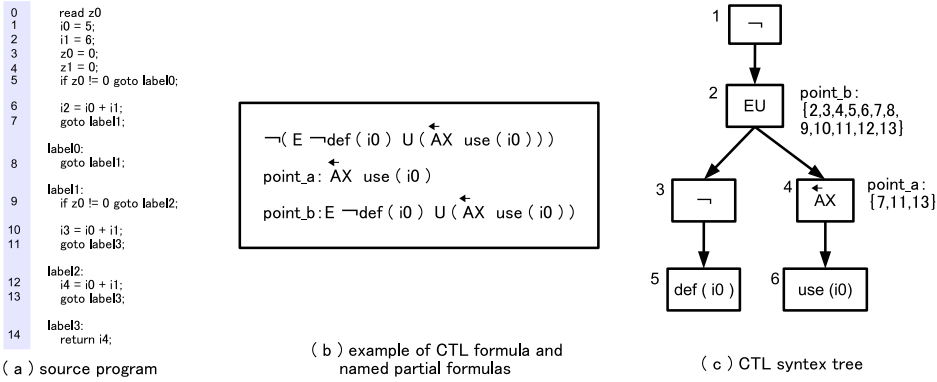

Fig. 6. Model checking

8 Experiments

Our experimental data were acquired by using the seven benchmarks of SPECjvm98 [15] and Okumura's Java code [12].

Experimental Environment

The experimental environment is as follows.

CPU: Celeron 2 GHz, Memory: 512 MB

Soot: version 2.2.0, JDK version: 1.5.0_06-b05

JVM options: -Xint -Xms128m -Xmx128m (to exclude the influence of JIT and the memory)

Optimization applied by our system: partial redundancy elimination (includes common subexpression elimination and loop invariant code motion), copy propagation (includes constant propagation), dead code elimination.

Future tense CTL is used in dead code elimination and partial redundancy elimination, but past tense CTL is used more often for most optimizations.

Optimization applied by Soot (for comparison): common subexpression elimination, partial redundancy elimination, copy propagation, constant propagation and folding, conditional branch folding, dead assignment elimination, unreachable code elimination, unconditional branch folding, and unused local elimination.

Processing Time of Optimization

The optimization time for the SPECjvm98 benchmark by our optimizer is shown in Table 3, and the optimization time for Okumura's Java code is indicated in Table 4.

code name	binding	model checking	rewrite	other	total
200_check	140	2138	48	939	3265
201_compress	328	4326	32	1236	5922
202_jess	296	13027	125	3624	17072
209_db	158	2687	31	1095	3971
213_javac	529	24797	46	6592	31964
227_mtrt	264	18080	93	1810	20247
228_jack	499	236069	202	3363	240133

Table 3
The optimization time of the SPECjvm98 benchmark (unit: millisecond)

code name	binding	model checking	rewrite	other	total
PiByMachin	125	344	16	46	531
CubeRoot	94	408	15	170	687
Cardano	110	1062	31	109	1312
CountingSort	125	468	15	79	687
NQueens	110	656	16	93	875
Jacobi	171	3690	32	482	4375
LogE	109	2362	48	263	2782
Fibonacci	109	344	16	187	656
Exp	141	1439	32	327	1939

Table 4
The optimization time of Okumura's Java code (unit: millisecond)

We see that the optimization time of our system is from 4 seconds to 4 minutes in seven of the benchmarks. It is slow compared to common compilers that take from milliseconds to several seconds. However, optimizations using temporal logic are generally slow because of traversal and searches on all instructions, and there

seems to be no other choice for the moment. Nonetheless, it is quite fast compared with results of previous work (some of which cannot perform such optimizations) [9] [2] [19].

Comparison of Execution Time

Figure 7 and Figure 8 show the comparison of execution times of the object code before and after optimization by our system and Soot (execution time without optimization is normalized to 1). Optimization by our system has a modest effect, although our optimization implements only a part of the optimization applied in Soot, and there are a few programs where our technique beat Soot. What influences the effect of optimization will be discussed in section 9.1.

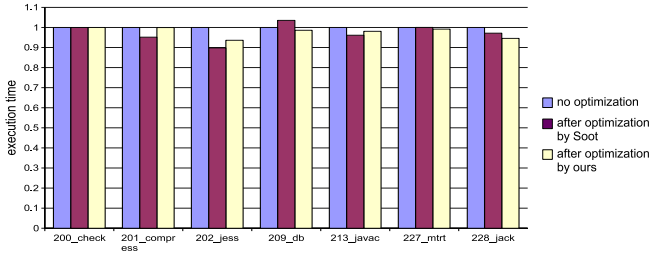

Fig. 7. Comparition of Execution times of SPECjvm98 benchmark

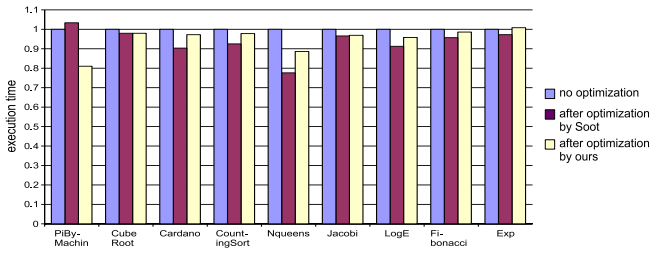

Fig. 8. Comparition of Execution times of Okumura's Java code

No previous work, such as [9] and Yamaoka [19], give the optimization time and the execution time data with the benchmarks, so we cannot make comparisons with previous studies.

Ban's system [2] allows past temporal logic operations written in PCTL [10]. It is transformed to those without past operations before model checking. So, we compared our system, which uses the same optimization formula of copy propagation involving past operations, with Ban's method. Figure 9 shows the result of the comparison we reran on the same machine as described before (Ban's optimization time is normalized to 1).

Example of Optimization Time Explosion Caused by Free Variables

Figure 10 shows an example of optimization time explosion caused by free variables. We described the specification of copy propagation as an example using different CTL_{bd} formulas, one with 2 free variables (left bar) and the other with 4 free variables (right bar). Optimization time increased from 4 seconds to about 39

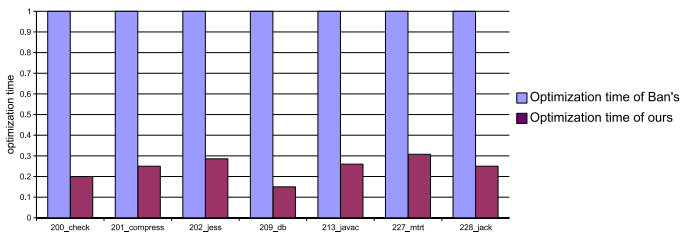

Fig. 9. Optimization time of our system compared with Ban's work

minutes , showing the dramatic increase in computational cost when the number of free variables is increased by two.

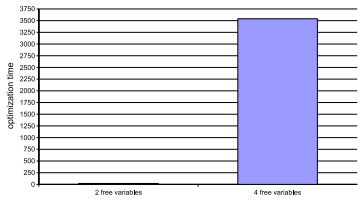

Fig. 10. Example of optimization time explosion (Vertical axis: second)

9 Discussion and Future Work

To our knowledge, our system is the first system that can make optimizers for real Java programs from specifications in CTL by using a model checker. Our main purpose is to clarify the possibility and problems of this approach and to consider how to overcome the problems. Consequently, there are many items for consideration about the current system in this section.

9.1 Consideration and Discussion

Expressive Power of CTL

Optimization can be specified easily and concisely in several lines by CTL_{bd}, but the expressive power of CTL_{bd} formula is inferior to common optimization algorithms in some cases. An example is when the algorithm cannot be represented by first-order logic such as conditional constant propagation [18]. Another example is proving the time-optimal property of partial redundancy elimination; it will be difficult without using the properties of dataflow equations.

Efficiency of The Compiler Optimizer

Binding of free variables greatly influences the efficiency of the system as mentioned in section 6. Moreover, model checking is an exhaustive algorithm. Therefore, its efficiency is inferior to that of common algorithm-based compiler optimizers.

Effectiveness of Optimization

Because our target is Java programs, instructions cannot be moved past an exception point. Instructions that may cause run-time exceptions like array index-

ing, division and the mod (remainder) operation etc. must all be excluded from optimization too.

The problem of obstruction of optimization by exceptions is known to be a common problem in Java optimizers.

9.2 Future Work

This section presents some possible future directions of research. The possibilities can be divided into two categories: improving optimization time and improving the efficiency of optimized programs.

Reducing Optimization Time

Free variables need to be reduced or eliminated as much as possible because binding of free variables greatly affects processing (optimization) time. The following program is an example illustrating the reduction of free variable binding and model checking.

1: $x := 100$;

...

5: $x := w + 1$;

6: $z := x + y$;

- Free variable binding and model checking can be omitted where it is not necessary. For example, if the target of copy propagation is x in "6: $z := x + y$", instructions prior to statement 5, such as statement 1, need not be checked.
- Binding can be omitted if the target is not on the path related to the temporal formula. For example, checking instructions on the past paths can be omitted if the CTL_{bd} formula includes only future temporal operators. Similarly, checking instructions far away from the next instruction can also be omitted if the temporal operator is AX or EX. In our experiment, when this technique is applied to the dead code elimination (which only includes future temporal operators), processing time is reduced to about 1/3.

  It will be also possible to introduce binary decision diagrams (BDD), partial evaluation or some other technology to make model checking faster.

Improving the Efficiency of the Optimized Program

To improve the effectiveness of optimization, overcoming the obstruction of optimization caused by exceptions and detailed analysis of loops, for statements, goto statements, etc., will be necessary.

Specifying complex optimizations such as conditional constant propagation [18] with CTL_{bd} is also our future work.

We are planning to carry out the above in the future.

10 Related Work

The crucial connection between model checking and program analysis was made by Cousot et al. [3].

Lacey et al. [7] introduced a temporal logic named CTL-FV that can use free variables in predicates. The papers [9] [8] describe the proposal and prove the correctness of some traditional optimization formulas by hand. The thesis [8] describes the detail of the technique. However, as for implementation, it just explains that they solved the problem by finding the fixed point after converting it into the μ-calculus. Moreover, the formulas that have been proven in the thesis are only a part of the optimization that can be done by real-world optimizers and no experimental data such as the optimization time are given.

Yamaoka et al. [19] have implemented an optimizer with CTL using the existing model checker SMV [14], but it can only deal with dead code elimination because only future temporal operators are allowed.

Ban's research [2] is able to treat the PCTL [10] including the past temporal operator in a limited form, by using 12 conversion equations, but it consumes time to remove past temporal operators, and the formulas become very long after conversion. As a result, the model checking time is considerable. Moreover, only the dead code elimination and copy propagation can be done because it can rewrite only one instruction corresponding to one condition in the optimization specification.

Lerner et al. [11] invented a domain specific language for writing compiler optimizations that can be automatically proved sound. However, their approach is different from ours based on temporal logic.

Experimental data on optimization time and execution time of optimized code using the benchmarks are not presented in any previous work [9] [2] [19].

11 Conclusion

The main contribution of our research is as follows.

- We implemented an efficient model checker that directly handles the CTL_{bd} without any conversion.
- We proposed an optimization specification language that is very expressive. As a result, even complex optimization formulas that can deal with real-life optimization can be written very naturally and easily.
- We developed a practical Java optimizer using CTL_{bd} that has a modest effect.
- It is the first time that experimental data on optimization time and the effect of optimization has been measured using the benchmarks and the test programs in our research. Most optimizations are approaching practicality.
- Moreover, some problems of optimization by CTL were clarified through this experience.

We think that these data and experiences are significant and valuable in this field and can contribute to future research work.

We plan to solve the existing problems and improve the performance toward a more realistic optimizer.

References

[1] Aho, A. V., Lam, M. S., Ravi, S. and Ullman, J. D.: Compilers Principles, Techniques, and Tools. Second ed., Addison Wesley, 2006.

[2] Ban, N., Hu, Z. and Takeichi, M.: Declarative description of Java program optimization and its efficient implementation (in Japanese), *Proc. 6th Programming and Programming Language Workshop (PPL2004)*, pp. 65–75, 2004.

[3] Cousot, P. and Cousot, R.: Automatic synthesis of optimal invariant assertions: mathematical foundations, *SIGPLAN Not.*, Vol. 12, No. 8, pp. 1–12, 1977.

[4] Clarke, E. M. Jr., Grumberg, O. and Reled, D. A.: Model Checking. The MIT Press, 1999.

[5] Fang, L. and Sassa, M.: Java optimizer with bidirectional CTL. Research Report C-230, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2006.

[6] Van Leeuwen, J. (Ed.): Handbook of Theoretical Computer Science Vol. B, Formal Models and Semantics, Elseveir Science Publishers B. V., 1990.

[7] Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Proving correctness of compiler optimizations by temporal logic. *In Proceedings of Symposium on Principles of Programming Languages*, pp. 283–294, 2002.

[8] Lacey, D.: Program transformation using temporal logic specifications, PhD Thesis, University of Oxford, 2003.

[9] Lacey, D., Jones, N. D., Van Wyk, E. and Frederiksen, C. C.: Compiler optimization correctness by temporal logic. *Higher-Order and Symbolic Computation*, Vol. 17, No. 3, pp. 173–206, 2004.

[10] Laroussinie, F. and Schnoebelen, P.: Specification in CTL+Past for verification in CTL. *Information and Computation*, Vol. 156, No. 1/2, pp. 236–263, 2000.

[11] Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automatically proving the correctness of compiler optimizations, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* , pp. 364–377, 2005.

[12] Okumura, H.: Algorithm dictionary by Java. http://oku.edu.mie-u.ac.jp/~okumura/Java-algo/

[13] Paleri, V. K., Srikant, Y. N. and Shankar, P.: A Simple algorithm for partial redundancy elimination. *ACM SIGPLAN Not.*, Vol. 33, No. 12, pp. 35–43, 1998.

[14] SMV Model Checker, http://www.cs.cmu.edu/modelcheck/smv.html

[15] SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98

[16] Steffen, B.: Generating data flow analysis algorithms from modal specifications, *Science of Computer Programming*, Vol. 21, No. 2, pp. 115–139, 1993.

[17] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot — a Java optimization framework, *In Proceedings of CASCON 1999*, pp. 125–135, 1999, http://www.sable.mcgill.ca/soot/

[18] Wegman, M. N. and Zadeck, F. K.: Constant propagation with conditional branches, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2, pp. 181–210, 1991.

[19] Yamaoka, Y., Hu, Z., Takeichi, M., Ogawa, M.: Generation of program analyzer based on model checking (in Japanese), *IPSJ Transactions on Programming*, Vol. 44, No. SIG13 (PRO18), pp. 25–37, 2003.