Optimization Verifying Compilers

SATO Hiroyuki

Information Technology Center/Department of Frontier Informatics,
The University of Tokyo.

E-mail: schuko@cc.u-tokyo.ac.jp

***Abstract* — Today, compilers play a role of bridge between high/abstract level specification of computation/algorithm and low/concrete level of target architectures. They are responsible for both effectively implementing high level programming concepts and exploiting very concrete high performance engines, where optimizations are very crucial in every respect. However, the growing complexity of modern optimizations causes the proof of correctness complicated, and also makes guarantee of performance optimality almost impossible. In this paper, we propose optimization verifying compilers. They automatically prove the correctness of optimizations. They also guarantee the optimality of generated code in performance.**

***Keywords* — Compiler Optimization, Verification, Program Analysis, Performance Model.**

I. Introduction

Today, compilers play a role of bridge between high/abstract level specification of computation/algorithm and low/concrete level of target architectures. Since programming languages have been a central issue in computer science, many useful and complicated concepts have been coined. As the result, modern programming languages can represent abstract and complicated concepts such as type, objects and aspects. Effective implementation of such concepts is left to compilers. On the other hand, the complexity of modern computer architectures is continuously growing. Exploiting high performance engines in modern CPUs and systems built on them such as parallel computers is also left to compilers. The gap between programming languages and architectures is unrecoverable unless we have appropriate remedies. In other words, modern compilers are responsible for both effectively implementing abstract concepts and exploiting very concrete high performance engines.

In modern compilers, optimizers occupy the central position in every respect. Optimizers are a set of program transformers that are expected to show performance gain while preserving the semantics of programs. In the study of compilers, many optimization techniques have been investigated. Peep-hole optimizations are no longer attractive to researchers because their effects are very much limited. Instead, systematic optimizations through program analysis are today's concern. They include partial redundancy elimination, global value numbering and code hoisting, some of which are based on problem description by using multiple dataflow equations. As the consequence, the optimization description has become much complicated.

With larger complexity of description of optimizations, their effect has become harder to analyze. Optimizations must be discussed in terms of performance improvement together with the correctness of their application. However, we cannot be optimistic about it. As for the performance, it is well known that two optimizations can interfere with each other. For example, aggressive register allocation can make the register pressure higher, and it can cause other optimization unavailable. Up to now, heuristics is the only solution. As for the correctness of optimization, for example, there has been, and will be reported a long lst of bugs of GNU-C. In theory, an optimization is beyond a simple term rewriting system. Conditions for applying optimizations are so complicated that term rewriting theory does not suffice for the analysis of optimizations.

Therefore, we need formalism. Formalism is an indispensable tool for us first to analyze optimizations, and second to write a proof of correctness on it, and finally to make some guarantee of performance optimality.

In this paper, we propose **optimization verifying compilers**. Optimization verifying compilers guarantee that codes are generated through correct optimizations, and hence that they are correct in the sense that the semantics of optimized codes and that of unoptimized codes are the same. Furthermore, they also guarantee that the generated codes are optimal in performance. In these two points, optimization verifying compilers are based on a formal system. The verifications of correctness and performance optimality are done on the formal base. Of course, because the verifications for all programs are infeasible in theory, we must restrict our target to an appropriate class of programs. However, the class is expected to be general enough, and we believe that optimization verifying compilers are attractive both in theory and in practice. We also believe that they are essential for building safe and secure society empowered by advanced electronics.

The rest of this paper is organized as: Section II is devoted to the definitions and outline description of optimization verifying compilers. In Section III, we discuss the correctness of optimization. In Section IV, we study formal performance model and the way of guaranteeing optimality. In Section V, we survey related work. In Section VI, we give a summary of this paper.

II. Optimization Verifying Compilers

Optimization verifying compilers are compilers that automatically verify two essential requirements to optimizers: correctness of the program transformation and guarantee of performance improvement.

Our project is inspired by "verifying compilers" of Hoare [4]. Let us first review verifying compilers.

A. Hoare's Verifying Compilers

We summarize the outline of verifying compilers in Fig. 1.

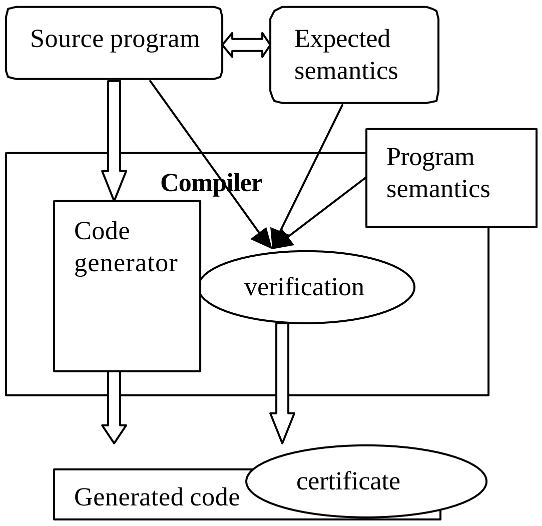

Fig. 1. Outline of Verifying Compilers.

Hoare (implicitly or explicitly) expects that a program is given with expected semantics, in other words, specification. A verifying compiler is expected to prove the correctness of a program in terms of its associated specification. As its result, a generated code is augmented with some proof of correctness (certificate). The idea of verifying a code for safe execution has become famous since Java processors implemented a verifier for incoming Java bytecodes. Giving a certificate/proof is its natural extension. One point to be noted is that the verification must be done with respect to some fixed program semantics. Java's success partly owes to the simplicity of the program semantics: it can check just out-of-memory references. If the semantics reflects usual mathematics, proofs become much complicated. How to manage this complexity is really challenging.

B. Optimization Verifying Compilers

Our optimization compilers inherit much of Hoare's verifying compilers. Our focus is put on optimizers inside compilers. For optimizers, because preserving semantics and guaranteeing performance improvement after transformations are two essential requirements, we concentrate on "verifying" these two features.

Preserving Semantics

In our project, we do not care about a specification of a given program. Instead, we identify the semantics as the original program. Besides this simplification, we consider the framework of program semantics as an ordinary operational semantics. Therefore, what concerns is that an optimizer does not give any change with respect to the operational semantics. The proof is divided into two classes. One is to prove that an original code and the transformed code have the same semantics, and the other is to give a proof that an optimizer does correct transformation. Both kinds of verifications are our concern (summarized in the left half of Fig. 2).

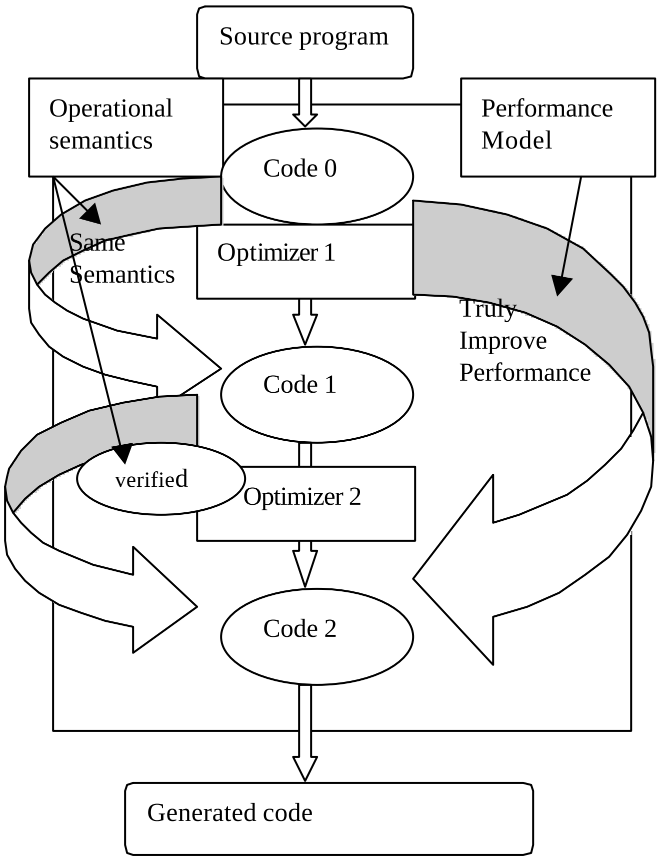

Fig. 2. Outline of Optimization Verifying Compilers

Performance Improvement

Performance is essential for evaluating optimizers especially for architecture oriented compiler writers. However, because the complexity of modern CPU architecture is soaring, it is a hard job to guarantee that an optimization really improves performance. Therefore, we must build an effective performance model so as to give a proof that a transformed code really improves performance. Moreover, the model must be discussed to enable discussions of features of optimizations. For example, if we can prove that a code is the best in performance, we do not need any further optimization. Conventional performance models based on simulations that are essential tools for designing CPUs are not enough. Performance verifying optimizers are shown in the right half of Fig. 2.

III. TYPE SYSTEM FOR PROVING CORRECTNESS

For proving that a given transformation preserves semantics, many systems and logics have been proposed. Among those, model checking and temporal logic are major systems for manipulating the change of states that exactly represents operational semantics.

In our verification system, we build a type system for analyzing operational semantics. Type theory has originally been developed for analyzing the correspondence between mathematical logic (proof) and program execution. Recently, it is proved to be a powerful tool for program analysis, and many concepts of resource usage are analyzed as types. In this section, we review [10]. In our system, the target of analysis is "assignment, " which, together with jumps, is a major behavior of program execution.

A. Type System Definition

First, we define our simplified assembly language:

Definition. We define our assembly language as in Table 1. A program execution is described as change of values of finite variables.

Table 1. An Assembly Language

$i \in$ Integers	
x0,x1,…,y0,y1,… $\in$ Vars	
l $\in$ Labels	
v ::= x | i	(Value)
ins ::= aop|mov|jump	(Instruction)
aop ::= **add** x,y,v|**sub** x,y,v|**mul** x,y,v|**div** x,y,v| **mod** x,y,v	
mov ::= **mov** x,v|**ssa** x0,{l1:x1, l2:x2}	
jump ::= **beq** x,v,l|**bg** x,v,l|**bge** x,v,l|**bl** x,v,l|…|**jmp** l **ret** x	
I::= ϕ| I;ins	(Instruction Sequence)
B::= I|I;jump	(Basic Block)
P::= l1:B1;l2:B2;…ln:Bn	(Program)

We restrict our attention to SSA forms. Variables are represented by its name together with suffices. There is at most one assignment given to a given variable. To represent multiple assignment to a given variable in a program execution, a ϕ assignment is defined at join points of program execution. In our assembly language, **ssa** instruction in Table 1 corresponds to ϕ. In our system, variables providing values are given with labels from which control comes. This is essential in giving its dynamic semantics as operational semantics.

Definition. Dynamic semantics is given as the change of states. A state is defined as a pair of program counter and a function mapping variables to values. A special state HALT is given. When ret instruction is executed, the state falls into HALT.

We define our type system for optimization verifying compilers.

Definition. We define our type system as in Table 2. Type inference rules are given in Table 3.

Table 2. Types

Types
$\alpha \in$ Type Vars
τ ::= α|T|int(i)|
(τ0, τ1)+\|(τ0, τ1)-\|(τ0, τ1)*|
(τ0, τ1)/\|(τ0, τ1)%|
{L0: τ0, L1: τ1}| $\mu\alpha$.{L0: τ0, L1: τ1}
Environment
Γ ::= ϕ| Γ, x: τ

The type int(i) is the singleton type representing the singleton set {i}. For each arithmetic operation, there is defined a corresponding type. For the **ssa** instruction, there is defined an SSA type representing join of values. Finally, μ-type is defined. A type variable α is μ-bound in type $\mu\alpha$.{L0: τ0, L1: τ1}.

As a special type environment, we define Γ0 as types for 0-th variables. In other words, Γ0 specifies types of initial values. Using typing rules in Table 3, there is uniquely defined Γ0 for a given program.

Type inference is classified into two categories: \|- Γ for valid type environment, and Γ P\|- x: τ for valid type inference for a variable in a given program P. The definition is a usual one except the rule (cut). (cut) corresponds to substitution of types. Actually, (cut) enables multiple type inference for a variable. To make the resultant type unique, we define the (decidable) type equality in the next Definition.

Table 3. Typing Rules

(Valid Type Environment)

$\Rightarrow \ \vdash$

$\vdash \Gamma \quad \text{not } x \in \mathrm{Dom}(\Gamma) \quad \alpha \text{ fresh} \Rightarrow \ \vdash \Gamma, x{:}\ \alpha$

(Valid Type Inference)

$\vdash \Gamma \Rightarrow \Gamma \ P \vdash i{:}\mathrm{int}(i)$

$\vdash \Gamma \quad x{:}\ \tau \in \Gamma \Rightarrow \ \Gamma \ P \vdash x{:}\ \tau$

aop $x,y,v \in P \quad \Gamma \ P \vdash y{:}\ \tau 1,\ v{:}\ \tau 2$

$\Rightarrow \Gamma \ P \vdash x{:}(\tau 1, \tau 2)\mathrm{aop}$

(for each aop **add**, **sub**, **mul**, **div**, **mod**)

mov $x,v \in P \quad \Gamma \ P \vdash v{:}\ \tau \ \Rightarrow \Gamma \ P \vdash x{:}\ \tau$

ssa $x,(L0{:}x1,L1{:}x2) \in P \quad \Gamma \ P \vdash x1{:}\ \tau 1,\ x2{:}\ \tau 2$

$\Rightarrow \Gamma \ P \vdash x{:}\{L0{:}\ \tau 1, L1{:}\ \tau 2\}$

$\Gamma, x{:}\ \alpha, \Delta \ P \vdash x{:}\ \{L0{:}\ \tau 1, L1{:}\ \tau 2\}$

$\Rightarrow \Gamma, \Delta \ P \vdash \ x{:}\ \mu\alpha.\{L0{:}\ \tau 1, L1{:}\ \tau 2\}$

$\Gamma \ P \vdash x{:}\ \tau 1 \quad \Delta, x{:}\ \alpha, \Theta \ P \vdash y{:}\ \tau 2$

$\Rightarrow \Gamma + (\Delta, \Theta) \ P \vdash \ \tau 2[\tau 1/\alpha]$

(cut)

Definition. Let a type τ be given. We define its regular tree $T(\tau)$ as:

$T((\tau 0, \tau 1)op) = (op)\{T(\tau 0), T(\tau 1)\}$

$T(\mathrm{int}(i)) = (\mathrm{int}(i))$

$T(\mu\alpha.\tau) = T(\tau)[\mu\alpha.\tau/\alpha]$

Then, we define $\tau = \sigma$ as $T(\tau) = T(\sigma)$.

The next theorem validates this definition.

Theorem. For a program P and a type environment Γ, if $\Gamma P \vdash x{:}\ \tau$ and $\Gamma P \vdash x{:}\sigma$, then $\tau = \sigma$.

Example. Let a program P be given as:

```
L0: mov x0, 0; L1: ssa x1, (L0:x0, L1, x2);
add x2,x1,1; bl x2,100,L1.
```

Under the type environment $\Gamma 0 = \phi$, we have $\Gamma 0 \ P \vdash \ x1{:}\ \mu\alpha.\{L0{:}\mathrm{int}(0),\ L1{:}(\alpha, \mathrm{int}(1))+\}$. Actually, we have other inferences of the type of x0 such as $\{L0{:}\mathrm{int}(0), \mu\alpha.\{L0{:}\mathrm{int}(0),(\alpha,\mathrm{int}(1))+\}\}$, but they are all the same.

B. *Analysis of Optimizations using our Type System.*

Definition. Given programs P and Q, we define $P \sim Q$ as P and Q are bisimilar in state change.

To state soundness of our system, we need structural equivalence. Here, a program is understood as a network of blocks. At the end of a block, there is a jump instruction.

Definition. Two programs P and Q are structurally equivalent if they have bijective correspondence as a network of blocks, and furthermore, the last jump instructions in corresponding blocks are the same.

Although structural equivalence looks restrictive, we see almost all global optimizations preserve structures of programs, and therefore two programs before and after optimizations are structurally equivalent.

We define CV(P) for a program P as variables that appear in jump instructions in P except **ret**. Furthermore, we call variables live if they are concern of analysis. We define LV(P) for P as a set of live variables. Note that an arbitrary variable can be live.

Our soundness theorem is stated as follows:

Theorem (soundness). Let structurally equivalent programs P and Q be given. Let CV(P) = CV(Q) and LV(P) = LV(Q). Let $\Gamma 0$ for P and Q be denoted as $P\Gamma 0$ and $Q\Gamma 0$ respectively. We assume that if for each variable $x \in CV(P) \cup LV(P)$: $P\Gamma 0 \ P \vdash \ x{:}\ \tau 0$ and $Q\Gamma 0 \ Q \vdash \ x{:}\ \tau 1$, then we have $\tau 0 = \tau 1$. Then $P \sim Q$.

The soundness theorem is the basis of verifying correctness of optimizations. As its immediate consequence, we can verify dead code elimination.

Corollary. Let Q be P whose dead codes are eliminated. Then, $P \sim Q$.

To prove the correctness of other optimizations that can change assignment types such as constant folding, we must extend the soundness theorem to that some order on types is included.

IV. Guarantee of Performance Improvement

Our second goal is to guarantee performance improvement. Because of the complexity of modern CPU architectures, to prove that an optimization really improves performance is a hard job. First, we need precise performance model. Conventional simulation-based models are prohibitive because of high computational cost for calculating performance. On the other hand, old simple models are not sufficient if they cannot explain performance gain. Moreover, if an optimization is parametric in some factors, we need a symbolic model to enable a symbolic analysis. Furthermore, if an optimization allows multiple heuristics, we need to evaluate each heuristics, or to give the best solution if we do not like to be bothered by heuristics.

In this section, we discuss our preliminary results on guarantee of performance improvement.

A. Precise Performance Model

We need a precise performance model that can explain performance gain as the result of exploiting high performance engines of modern CPU architectures. Here, we take loop unrolling as an example of performance-sensitive optimization [18]. Today, the effect of loop unrolling is explained by instruction level parallelism (ILP), not by loop overhead reduction. Therefore, we need a precise ILP model.

Definition (unrolling shape). Let a loop be given. Among instructions of the loop, we only consider memory operations and arithmetic operations, and omit loop control/jump operations. The set of the instructions are given as Ins0 + $\Delta \cdot n$ for unrolling factor n. We schedule them according to delay and throughput of each instruction. We call the constructed diagram "unrolling shape."

Note that unrolling shapes are simplified scheduling diagram of instructions, but that they are easy to compute, and are accurate in the sense of performance prediction. Actually, from the shape, we can compute the cycles taken by the loop as the function of unrolling factor. Our experience shows that unrolling shapes effectively work. Fig. 3 shows our experiment for a simple loop on SPARC. We can see that predicting the effect of loop unrolling is successful.

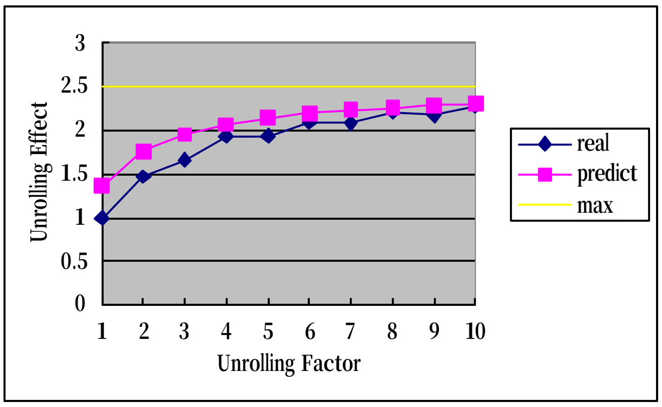

Fig. 3. Prediction of Effect of Loop Unrolling for a Simple Loop by Using Unrolling Shape.

Unrolling shape is originally studied for in-order processors. However, we have found that it can be applied also to out-of-order processors. On Alpha, we can construct a predictive performance model of loop unrolling using unrolling shape [15].

B. Guarantee of Performance Improvement

If an accurate performance model is given, the next job is to find a solution that is definitely better than the original code. Traditionally, because most architecture-oriented optimizations are computationally hard to find the optimum solution, heuristics must be adopted to obtain a solution in practical time. Performance-critical optimizations such as register allocation, instruction scheduling, and data layout are all NP-complete problems. Good heuristics are intensively studied to obtain near-optimal solutions in practical time. However, the strategy of heuristics has the essential flaw: it gives no guarantee of performance improvement, nor it says anything about the quality of solutions. Therefore, a given heuristics is evaluated for some arbitrary collected benchmark programs. Comparison between heuristic methods is inevitably comparative.

Recent soaring of computer performance gives an opportunity: finding the best solution for problems of reasonable size in reasonable time is no more necessarily impossible. If we can find the best solution, we can also evaluate heuristics with respect to it. In other words, we can have an absolute evaluation.

Problem formulation using integer programming is a promising direction in this sense. In our group, we have succeeded in defining a performance model of Pentium4 considering memory hierarchy of register/L1 cache/L2 cache/memory, and behaviors of instructions taking memory as operands in the framework of integer framework [5]. Using this model, we have improved the register allocation of GCC 3.3, and the performance has been improved up to 2%. This also indicates that the register allocation heuristics of GCC 3.3 is quite good.

V. Related Work

The term of "verifying compiler" was coined by Hoare [4]. Much of our project is inspired by his project. His "verifying compiler" aims at verifying the correctness of a program at compile time. It is a very aggressive, but challenging project. We put focus on verifying two most significant requirements of optimizers.

There are many formal systems proposed for analyzing optimizations [7,8,13]. Translation validation [13,14,19] is proved powerful for verifying the correctness. The strategy of translation validation is to check whether two programs are bisimilar on simple operational semantics. Lacey [7] has applied temporal logic to express the condition under which an optimization can safely be applied. Applying type system to program analysis on low-level languages has started in TAL project [11,12]. The type system of TAL can express that an instruction is safe to execute in the sense that it does not access any out-of-memory region. Matsuno [9] extends it to allow arithmetic in calculating an address.

Accuracy of performance model is strongly required especially to large programs. Kerbyson [6] aims at modeling patterns of communications and computation, and has a predictive model. On an accurate model, we can build a firm base of optimum optimization. In particular, the Integer Programming method provides powerful framework. Many problems such as register allocation [1,3], instruction scheduling [17] and data layout [2] are formalized by using Integer Programming. The remaining work is to solve the problem to obtain the best solution. Adaptive compilation and auto-tuning [16] are variants for finding the best solution. However, they are just accumulated know-how, and do not need any science.

VI. CONCLUDING REMARKS

In this paper, we have presented optimization verifying compilers. First, we have explained the grand design of our project. In Section III, we have studied the type system on which we can verify semantics-preservation. In Section IV, we have discussed the accuracy of performance model, and guarantee of performance improvement by obtaining the optimum solution using the integer programming. Our project aims at assuring security of processors of programming languages in very scientific way. Our primary concern is on theoretical side, but we believe that it will practically give a firm base for the future IT infrastructure.

REFERENCES

[1] Appel, A., George, L., "Optimal Spilling for CISC Machines with Few Registers," Proc. PLDI'01, 2001.

[2] Bixby, R., Kennedy, K., Kremer, U., "Automatic Data Layout Using 0-1 Integer Programming," Proc. PACT'94, 1994.

[3] Goodwin, D., Wilken, K., "Optimal and Near-optimal Global Register Allocation Using 0-1 Integer Programming," Software Practice and Experience, vol. 26, 1996, 929-965.

[4] Hoare, T., "The Verifying Compiler: A Grand Challenge for Computing Research," J. ACM vol. 50, 2003, 63-69.

[5] Horimoto, K., An Optimal Register Allocation for X86 Architecture with Consideration of Memory Hierarchy, Master Thesis of Department of Frontier Informatics, the University of Tokyo, 2004.

[6] Kerbyson, D.J., Alme, H.J., Hoisie, A., Pertinei, F., Wasserman, H.J., Gittings, M., "Predictive Performance and Scalability Modeling of a Large-Scale Application," SC2001, 2001.

[7] Lacey, D., Jones, N.D., van Wyk, E., Frederiksen, C., "Proving Correctness of Compiler Optimizations by Temporal Logic," Proc. POPL'02, 2002.

[8] Lerner, S., Millstein, T., Chambers, C., "Automatically Proving the Correctness of Compiler Optimizations," Proc. PLDI'03, 2003.

[9] Matsuno, Y., Sato, H., "Flow Analytic Type System for Array Bound Checks," ENTCS vol. 76, 2003.

[10] Matsuno, Y., Sato, H., "A Type System for Optimization Verifying Compiler," to appear in PPL2004.

[11] Morriset, G., Walker, D., Crary, K., Glew, N., "From System F to Typed Assembly Language," ACM Trans. Programming Lang. and Syst., vol. 21, 1999, 528-569.

[12] Necula, G., "Proof-carrying Code," Proc. POPL'97, 1997.

[13] Necula, G., "Translation Validation for an Optimizing Compilers," Proc. PLDI'00, 2000.

[14] Pnueli, A., Siegel, M., Singerman, E., "Translation Validation," Proc. TACAS'98, LNCS 1384, 1998.

[15] Sato, H., Yoshida, T., "Unrolling Shape: Symbolic and Quantitative Analysis of Loop Unrolling Effect," Proc. 6th Int'l Conf. Software Engineering and Applications, 2002.

[16] Whaley, R., Petitet, A., Dongarra, J., "Automated Empirical Optimizations of Software and the ATLAS Project," Parallel Computing, vol. 27, 2001, 3-25.

[17] Wilken, K., Liu, J., Hefferman, M., "Optimal Instruction Scheduling Using Integer Programming," Proc. PLDI'00, 2000.

[18] Yoshida, T., Sato, H., "Characteristics Extraction of Loop Unrolling and its Modeling," Trans. IPSJ vol. 42 (SIG 7(PRO11)), 2001, 1-11. (in Japanese)

[19] Zuck, L, Pnueli, A., Fang, Y., Goldberg, B., "VOC: A Translation Validator for Optimizing Compilers," ENTCS vol. 65, 2002.