# Automated Test Program Generation for an Industrial Optimizing Compiler

Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, Zhaohui Wang
Institute of Software, Chinese Academy of Sciences
P.O. BOX 8718, Beijing, China
zhaochen@iscas.ac.cn, yunzhi,qiuming@itechs.iscas.ac.cn
guoliang@pubmail.iscas.ac.cn, wzh@itechs.iscas.ac.cn

## Abstract

*This paper presents joint research and practice on automated test program generation for an industrial compiler, UniPhier, by Matsushita Electric Industrial Co., Ltd. (MEI) and Institute of Software, Chinese Academy of Sciences (ISCAS) since Sept. 2002. To meet the test requirements of MEI's engineers, we proposed an automated approach to produce test programs for UniPhier, and as a result we developed an integrated tool named JTT. Firstly, we show the script-driven test program generation process in JTT. Secondly, we show how to produce test programs automatically, based on a temporal-logic model of compiler optimizations, to guarantee the execution of optimizing modules under test during compilation. JTT has gained success in testing UniPhier: even after benchmark testing and comprehensive manual testing, JTT still found 6 new serious defects.*

## 1 Introduction

This paper shows research and practice on automated test program generation for an industrial compiler: a script-driven test program generation process, a formal model-based approach for generating test program for optimizing modules in any compiler, as well as a brief report of an industrial application.

Matsushita Electric Industrial Co., Ltd. (MEI) has developed an EC++ compiler, named UniPhier [12], for its new processor used in its embedded products. To improve its performance, a large number of compiler optimizations have been implemented in UniPhier. Obviously, UniPhier, especially the compiler optimizations, need to be fully tested to ensure its correctness.

Benchmarks and public test suites are often used to test compilers. Though they can provide a standard evaluation on a compiler's performance or correctness, they cannot be used to test compiler-specific features, like optimizations, data structures, compilation directives, and so on, especially when these features are interleaved. To test UniPhier thoroughly, MEI's test engineers have to prepare a large number of programs. But, in practice, the manual preparation of such test programs is labor-intensive, and test efficiency and test quality are difficult to achieve.

In September of 2002, MEI and the Institute of Software, Chinese Academy of Sciences (ISCAS) initiated a joint project for developing a testing tool to test UniPhier automatically. We have achieved significant improvement in testing UniPhier, and delivered an automated tool named JTT (Jade Testing Tool, where Jade Hotel is the location launching this project).

By setting several flexible parameters in a domain-oriented script, JTT can generate any required number of programs described by the parameters automatically. The script in JTT is designed to specify test engineers' requirements for test programs in detail. It provides test engineers with flexibility to produce arbitrary legal programs. In contrast, performance benchmarks, public test suites and some grammar-based generation approaches [2, 11, 8, 10, 4] can only provide fixed types of programs.

The process of test program generation in JTT is driven by a script that includes two levels: High Level Script (HLS) and Middle Level Script (MLS). HLS is a simple and abstract specification for parameters of the desired programs, such as compiler optimization under test, data types to be used, etc. MLS is a template containing some abstract statements in desired program and their relationships. HLS is mainly for test engineers, while MLS is an intermediate format used to support both translating from HLS to test pro-

*AST'09*, May 18–19, 2009, Vancouver, Canada
978-1-4244-3711-5/09/$25.00 © 2009 IEEE

grams and writing test programs for special test requirements. Test requirements are specified in HLS, which are then translated automatically by JTT into test programs. Final programs are produced according to the template in the MLS, and therefore obey the specification in the HLS.

A compiler optimization can be specified as a name in HLS, and the transformation process from a name to a program template in MLS is based on temporal logic. A temporal logic formula is used to describe action and prerequisites of optimizations, which help construct test programs automatically. The correctness of optimization specification by temporal logic is discussed in [9]. We chose CTL [9] to formally specify optimizations. Based on the formalization, we construct a Node Control Graph (*NCG*) which represents the semantics of the specification. A *NCG* is composed of nodes labeled with temporal logic formulas and edges labeled with data constraints. By expanding the formulas and satisfying the data constraints, a *NCG* is converted into a program template specified in MLS. Also, we propose an approach to produce loop-independent and loop-carried data dependence within loop nests, which are necessary for loop optimizations. The formal specification of compiler optimizations and the derived generation process are transparent to HLS and MLS scripts.

JTT has succeeded in testing UniPhier. Before using JTT, UniPhier had been tested using some popular benchmark, test suites and manually coded test programs. Even so, JTT still found six new serious defects, and improved average statement coverage of seven important optimizing modules in UniPhier from 37% to 72%.

This paper is organized as follows. Section 2 provides an overview of test program generation process in JTT. Then we discuss a temporal logic based test program generation approach for both scalar optimizations and loop optimizations in section 3. Section 4 reports experiences in testing UniPhier. Concluding remarks are made in section 5.

## 2 Automated Test Program Generation in JTT

JTT is composed of five components: test program generation, test data management, test execution, test result analysis, and system configuration. Test execution and test program generation are two key components of JTT. Test execution compiles test programs using the compiler under test and a trusted reference compiler (such as GCC). If both are successful then execute the binary code respectively and compare outputs. Test failure is reported when one of the compilations fails or the outputs are not same. This method is easy to automate and quite efficient. In this paper we focus on test program generation, which involves special techniques.

To generate test programs automatically, we design two levels of script languages: HLS and MLS. The former is provided to test engineers to specify test programs and the latter is used as an intermediate representation in the process of test program generation. Figure 1 shows the two-phase of test program generation: generation of MLS scripts according to HLS scripts and generation of test programs according to MLS scripts. The two phases are accomplished by two translators, H2M (HLS to MLS) and M2P (MLS to Program) respectively .


**Figure 1. The Process of Test Program Generation in JTT**

HLS provides parameters for users to customize and specify test programs, and many parameters have multiple possible values. Test engineers can specify values of these parameters through a GUI editor and the GUI editor will automatically produce HLS scripts (see Figure 2).

```
Element {
   Var-type: char, pointer(int), struct(char*pointer(int))
   Statement: if, for, switch, goto
   Operator: +, *, <<, &
}
Structure {
   Intra-module: Type = if+loop, Nest = 3, Parallel = 2
   Inter-module: Depth = 2, Width = 3, Recursion = 0
}
Content {
   ScalarOpt: DCE, CSE
   LoopOpt: Skew, Fusion, Interchange
   Pragma: Prefetch, UnrollLength
}
```

**Figure 2. An Example HLS Script**

HLS contains three parts: Element, Structure, and Content. Element part specifies basic elements of a test program such as variable types and operators that should be used. Variable types include both primitive types of the source language such as *int* and *char*, and compound types such as *array* and *pointer*.

Structure parts specify how many branch structures and loop structures are used in test programs. Content

part specifies contents related to optimizations. There are two kinds of optimizations to be tested: scalar optimizations that do no involve nested loops, and loop optimizations that involve nest loops. Besides these general optimizations, UniPhier also introduces a special set of optimizations directed by optimization directives (*pragma* directives). To test an optimization, test engineers only need to list its name in a HLS script.

A MLS script contains same parts as a HLS script except that the Structure part of a MLS script contains more details on test programs. The Structure part can specify how the desired branch statements and loop statements are arranged in test programs. For example, the parameter Times of LOOP of the example script in Figure 3 specifies the number of iterations in a loop. These parameters are transparent to test engineers, and are used to generate test programs for optimizations.

```
Element { ... }
Structure {
    Func1( ){
      IF(||,2;&&,3){
        ASS
      }
      LOOP(Times=20){
        Func2
        ASS
      }
    }
    Func2( ){
      ASS
    }
}
Content { ... }
```

**Figure 3. An Example MLS Script**

MLS scripts can be generated automatically by the translator H2M according to HLS scripts. H2M implements the following functions: 1) generating global control structures according to parameters in HLS scripts; 2) generating special local control structures, such as perfect nested loops, for optimizations, and inserting the local structures into the global structures; 3) copying all undetermined parameters of Element and Content parts of the HLS script into the corresponding parts of MLS scripts.

M2P generates test programs from MLS scripts. Inside M2P, quite a few fundamental functions for constructing a complete test program are implemented. All these functions fall into the following categories: 1) generating symbolic variables and maintaining the variable list; 2) generating various expressions and statements; 3) conducting some necessary pre-computation to ensure the generated programs can be executed without errors such as infinite loops; 4) generating and maintaining internal representations of test programs; 5) translating the test program's internal representations into real test programs.

Based on the above functions, M2P can generate complete test programs, but this is not enough for testing compiler optimizations. To test various optimizations, complicated statements need to be generated. For the optimizations directed by *pragma* directives, it is relatively straightforward, because relations among the statements to be generated are simple. For general optimizations, including scalar optimizations and loop optimizations, it is more difficult. We will discuss this in section 3.

## 3 Model-based Test Program Generation for Optimizing Modules

Model-based testing is "a kind of software testing in which test cases are derived in whole or in part from a model that describes the system under test" [3]. Modeling a large complex system, like an optimizing compiler, can be difficult. But modeling key components in such a large system, like optimizing modules in a compiler, is easier and more practical. In this section we show how to model compiler optimizations using temporal logic and how to produce test programs for both scalar and loop optimizations.

Effectively testing an optimization requires the execution of the corresponding optimizing modules when a test program is compiled. So the test programs must have the elements that an optimization manipulates and the prerequisites that the optimization can be safely applied. We call such programs focused ones for an optimization because they can guarantee the execution of the optimization module during compilation. Notably, the requirements for focused programs include both program structure and expression in statements. We use temporal logic to formalize such requirements and produce focused test programs based on the formal specifications.

Temporal-logic-based test program generation can be automated using the algorithms illustrated in this section. Moreover, it allows us to produce test programs that are hard to enumerate by hand. For instance, Dead Code Elimination (see section 3.1) involves an assignment to a variable $x$, another assignment to the same variable, and a reference to $x$. Each of the 2 or 3 statements can be located in a sequence block, a branch block, a loop block and even in an expression. Enumerating all possible combinations of different locations of the 2 or 3 statements by hand is difficult and some combinations may be omitted. But with the aid of model-based test program generation, it is easy to enumerate all combinations, which improves

test efficiency further.

The specification and derived generation process are transparent to HLS and MLS scripts. A test engineer only needs to specify the name of the optimization they would like to test in HLS with some necessary parameters, without considering any details of test program generation.

### 3.1 Overview of Temporal Logic based Generation

Computation Tree Logic (CTL) is a logic to describe a transition system. David Lacey first used CTL to formally specify scalar optimizations [9]. We have extended CTL to describe loop optimizations [15].

A transformation is denoted by a conditional rewriting rule $I \Rightarrow I'\ if\ \Phi$, where $I$ is a set of CTL formulas about instructions derived directly from the original program, $I'$ is the transformed version of $I$ after optimization, and $\Phi$ is called a side condition which is a CTL formula that explains why the optimization can be safely applied. For example, Dead Code Elimination (DCE) can be denote by follows:

$$\begin{aligned} & n : (x := e) \Rightarrow rm\_stmt(n) \\ if \quad & n \models \neg EX(E(\neg def(x)\ U\ use(x) \wedge \neg node(n))) \end{aligned}$$

Where $n$ is the label of a statement, $rm\_stmt()$ removes a statement from the original program. The above formula uses several logic quantifiers in CTL: E is a path quantifier, and X and U are temporal quantifiers. It specifies that DCE removes assignment statements that assign a never used value if there does not exist a path that can go through a node that uses $x$ without a different instruction re-assigning to $x$ first. More specifications of optimizations can be found in [9].

The specifications can be mapped to nodes in the control flow graph and edges labeled with formulas connecting them. These nodes and edges together show a simplified control flow graph that is a template for all desired programs. In the rest of this paper we name it Node Control Graph ($NCG$) for convenience. By satisfying the formulas at edges, $NCG$ can be converted into a program template in MLS. The process is illustrated in Figure 4.

In Figure 4, *normalization of side condition* transforms the original side condition equivalently for convenience of the construction of the $NCG$; *construction of* $NCG$ generates key nodes and edges; *construction of* $CFG$ enriches the complexity in resulting code by constructing a $CFG$ according to $NCG$; *expansion of temporal quantifiers* expands formulas on nodes in $CFG$ to all possible nodes along some computational paths according to the requirements of path quantifiers and

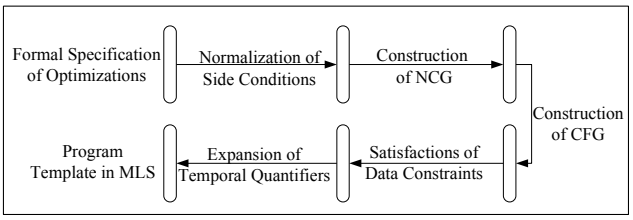



**Figure 4. Process of Translating a CTL Formula to Program Template**

state quantifiers; *satisfaction of data constraints* generates necessary expressions and places them in nodes to produce a program template. Figure 5 shows a step-by-step process for DCE from CTL formula to a program template in MLS.

Loop optimizations involve data dependence within loop nests, but the data dependence is related to the relative execution order of statements in loop nests, rather than the sequential order in a static CFG. Therefore, generating data dependence requires a special technique. Except for that, the techniques of generation test programs for scalar optimization and loop optimization is the same. We discuss test program generation for scalar optimizations in 3.2, and propose a special technique for generation of data dependencies in loop nests in 3.3.

Not only test programs for an optimizing module applies an optimization can be generated by this technology, but that for it does not apply an optimization can be generated by the same technology. In this paper, we focus on the former.

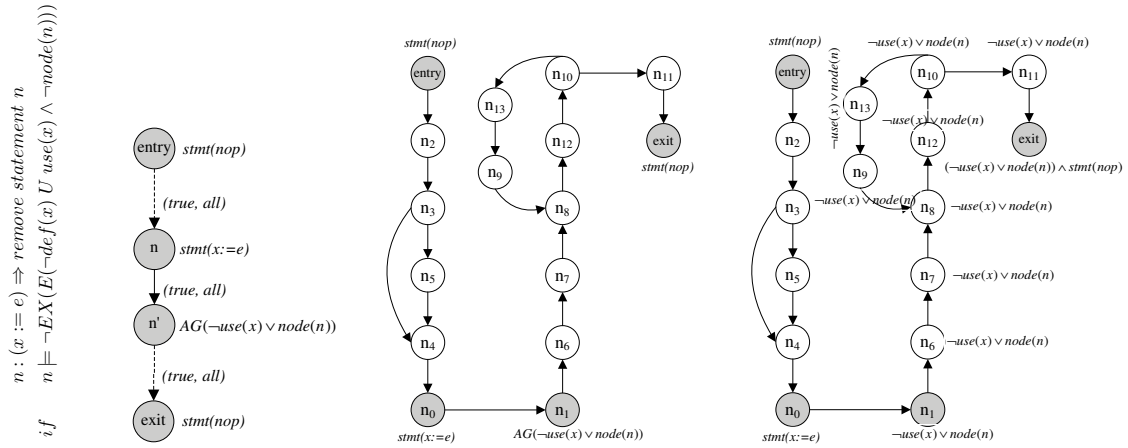
### 3.2 Test Program Generation For Scalar Optimizations

In the above steps, construction of the $CFG$ and satisfaction of data constraints are quite straightforward. The remaining steps are discussed below.

#### 3.2.1 Normalization of Side Conditions

It is difficult to generate test programs based on the original side conditions directly, for two reasons: 1) the $\neg$ operator prevents the generation since we cannot construct an object that does not "exist"; 2) multiple possibilities exist in some formulas which makes generation more complicated. Before the generation, we transform the original side conditions to simplify the generation.

We use 10 rules for equivalence transformations [15]. The rules can be iteratively employed to eliminate the




(a) DCE Formula (b) NCG of A Possible DCE Instance (c) CFG Labeled with Temporal-Logic Formula (d) Program Template

**Figure 5. Example Generation for DCE**

above cases. Generally these rules should be applied in the following order: $X$ operator, operator merge, $\neg$ operator, and then W operator.

There are some $\vee$ operators in the original side conditions specified as CTL formulas, which mean the conditions can be met when any sub-condition holds. After the equivalence transformation, some extra $\vee$ operators may be introduced. It is clear that multiple possibilities will make test program generation more difficult. We divide these sub-conditions into a set of single ones of which a single condition meets a single case by the rules in [15].

### 3.2.2 Construction of Node Control Graph

The following are some formal definitions about NCGs.

**Definition 1** A critical node is defined as $cn = (id, type, vp)$ where $id$ is a unique name in the basic block, $type$ denotes node type with $type \in \{Entry, Seq, PreLoop, LoopHead, LoopBreak, LoopTail, OutLoop, Branch, EndBranch, Exit, Undefined\}$, and $vp$ is a CTL formula associated with $cn$. All $cn$ nodes make a set $CN$.

**Definition 2** A property for a critical edge is defined as $\mathcal{P} = (\mathcal{F}, \mathcal{Q})$ where $\mathcal{F}$ is a CTL formula that should be met by nodes along a computational path. $\mathcal{Q} \in \{ALL, EXIST\}$. If $\mathcal{Q}$ = ALL, all nodes along all paths must meet $\mathcal{F}$. If $\mathcal{Q}$ = EXIST, all nodes along at least one path must meet $\mathcal{F}$.

**Definition 3** A critical edge is defined as $ce = (i, j, isreal, \mathcal{P})$, where $i$ is the source node and $j$ is the destination node of the edge, and $\mathcal{P}$ is defined as above, and $isreal \in \{TRUE, FALSE\}$. If $isreal$ = TRUE, this edge is called a *realedge*. It cannot be replaced by another edge and $\mathcal{P}$ does not play any role in this case. Otherwise, this edge is called a *virtualedge* and can be replaced by one or more real edges. In other words, there may be more than one node between the source node and the destination node of a virtual edge. All $ce$ edges make up a set $CE$.

**Definition 4** A node control graph $NCG$ is defined as a directed graph $NCG = (CN, CE)$, where $CN$ is defined as Definition 1, $CE$ as Definition 3.

According to Definition 4, a $NCG$ includes a set of critical nodes and a set of critical edges. We start the construction with an initialization for $CE$ and $CN$.

Initially, $CE$ and $CN$ are empty. Two critical nodes, Entry denoted by $(entry, Entry, stmt(skip))$ and Exit denoted by $(exit, Exit, stmt(skip))$ are added to $CN$. Then suppose there are $k$ nodes to be modified, denote by $n_0, n_1, ..., n_{k-1}$, and corresponding statements $stmt(s_i)(0 \leq i < k)$. For any $0 \leq i < k$, a critical node of the form $(n_i, undefined, stmt(s_i))$ is also inserted into $CN$.

Side conditions fall into two categories according to path quantifiers and state quantifiers. Without any quantifier, construction of $NCG$ is straightforward. With path quantifiers or state quantifiers, a condition involves at least two nodes in control flow graph. The following algorithm is to construct $NCG$ according to such a formula.

**Algorithm 1** The variable $c$ denotes a condition under processing, and the routine $quant(c)$ gives the quantifier specified in the condition, the routine $path\ (c)$ extracts the path quantifier where $path\ (c) \in (A,\ E,\ \overleftarrow{A},\ \overleftarrow{E})$, the routine $state(c)$ extracts state quantifier where $state(c_i) \in (G,\ F,\ none)$. Temporal

variable $p$ denotes the formula before the $U$ operator, and $q$ denotes the one just after the $U$ operator.

```
if quant(c) = none then
    cn(n) := (id(n), type(n), vp(n)∧p)
else
    CN := CN ∪ (c, undefined, q)
    if path(c_i) = A then
        e :=(n, c, false, (p,ALL))
    elseif path(c_i) = E then
        e :=(n, c, false, (p,EXIST))
    elseif path(c_i) = ←A then
        e :=(c, n, false, (p,ALL))
    elseif path(c_i) = ←E then
        e :=(c, n, false, (p,EXIST))
    endif
    CE := CE ∪ {e}
endif
```

So far, all the necessary information about why a given optimization can be safely applied is presented in the $NCG$. To make $NCG$ a strongly connected graph like a $CFG$, we need to connect the Entry node to a node whose incoming degree is 0, and connect the exit node to a node whose outgoing degree is 0.

#### 3.2.3 Expansion of Temporal Operators

So far, we have not covered the temporal logic formulas associated with nodes in the $NCG$. These formulas determine the property of more than one node in the $CFG$, but are associated with only a single node. So the formulas have to be expanded to accommodate all possible nodes along some computational paths according to the requirements of the path quantifiers and state quantifiers.

To achieve this, we construct a computational path from a node to the Exit node of the $CFG$. It is notable that a path of this kind differs from a computational path in CTL, which is a path of infinite length. To avoid any confusion, a computation path from a node to the Exit node is called Simple Computational Path ($SCP$). A $SCP$ is defined as $(n_0, n_1, ..., n_k, exit)$, where $n_0$ is any node but the Exit node, and for any $0 \leq i, j \leq k$, $n_i$ differs $n_j$. The difference means that for any cycle in CFG, nodes along this cycle appear only once in a $SCP$.

According to this definition, we can easily extract a $SCP$ by traversing a CFG in depth-first order and back-tracking when a node that is encountered a second time. While visiting a node, we can distribute the formula associated with it to all the nodes along a $SCP$ starting from it according to the path and temporal quantifiers. If path quantifiers and state quantifiers are $AG$, we find all subsequent nodes and attaches the required property to them. If the quantifiers are $AF$, $EG$ or $EF$, a random strategy is used to determine which paths and/or which nodes are selected to hold the required property.

### 3.3 Data Dependence Generation in Loop Nests

In UniPhier, many loop transformations have been introduced to improve performance, similar to those introduced in literature [7]. These loop transformations usually optimize nested loops based on data-dependence analysis [1], and most of them requires specialized characteristics of data dependency within loop nests.

There are three types of data dependence that affect the validity of loop transformations: flow-, anti-, and output- dependence [1], denoted by $\delta^f$, $\delta^a$, $\delta^o$, respectively. A data-dependence may also exist between two execution instances of statements within a single loop or nested loops. If the two instances are not inside the same iteration of a loop, the dependence is called a loop-carried dependence, otherwise it is called loop-independent dependence. Loop-carried dependence is usually represented by Dependence Distance Vector (DDV) [1] and their three basic types are denote by $\delta^{Vf}$, $\delta^{Va}$, and $\delta^{Vo}$ respectively in this paper.

To generate nested loops that satisfy specified data dependence, we use the Process Graph [13] as a model of nested loops. Process Graphs are suitable for representing program's internal data-dependence, and they are similar to the well-known model Program Dependence Graph [6] except that Process Graphs exclude programs containing unstructured jumps such as *goto* statements. A Process Graph is a directed graph G=(N, E), where N is the node set and E is the edge set. In N there are two special nodes, the start node $e$ and the end node $x$. Other nodes in N represent computation units, which can contain a single statement, a boolean expression, or a single-entry-single-exit module. Edges in E fall into two categories: solid edges and dashed edges, representing control-dependence and data-dependence respectively.

The generation process for loops with data-dependence consists of three phases: 1) Construct control structure of the Process Graph of nested loops; 2) Add valid data dependence relations between nodes in the Process Graph; 3) Fill in the nodes of the Process Graph with specific statements. Phase 2) and 3) are the two key phases that generate the data-dependence.

In phase 2), before adding data dependence, a set of valid DDVs that satisfy the formal specification in the

script should be generated. We implement it by an enumerative algorithm whose time complexity is high but acceptable because the scale of the problem is small.

In phase 3), dependence construction includes construction of loop-independent dependence and loop-carried dependence. The process is as follows.

(1) Determine the type of loop-carried dependence. Assuming there exists relation a $S_1\delta^V S_2$ between nodes $S_1$ and $S_2$. If there exists a node $S_3$ and relation $S_1\delta^o S_3$ holds, then the relation $\delta^V$ between $S_1$ and $S_2$ should be determined as $\delta^{Va}$. If relation $S_3\delta^o S_2$ holds, then the relation $\delta^V$ should be $\delta^{Vf}$. Otherwise we randomly determine the type of $\delta^V$.

(2) Determine all definition variables of each statement. If there exists a relation $\delta^o$ between two statements, then the two definition variables should be the same, otherwise should be different.

(3) Construct loop-independent dependence. Relation $\delta^o$ is considered but only $\delta^f$ and $\delta^a$ need to be handled. Assuming that the definition variable of $S_1$ is x and definition variable of $S_2$ is y, if $S_1\delta^f\ S_2$, add x to the reference list of $S_2$, and if $S_1\delta^a\ S_2$, add y to the reference list of $S_1$.

(4) Construct loop-carried dependence. Assuming that there is a loop-carried dependence between $S_1$ and $S_2$, the corresponding DDV is $[v_1,v_2,,v_n]$, and definition variable of $S_1$ is $A[a_1i_1+c_1][a_2i_2+c_2][a_ni_n+c_n]$. If the dependence is a flow-dependence, then we determine one reference item $A[b_1i_1+d_1][\ b_2i_2+d_2][b_ni_n+d_n]$ according to the equations $c_k = b_kv_k+d_k$; $a_k = b_k$; $(1 \le k \le n)$ and add the item to the reference list of $S_2$. If the dependence is an anti-dependence, then we determine the reference item according to the equations $d_k = a_kv_k+c_k$; $a_k= b_k$; $(1 \le k \le n)$ and add them to the reference list of $S_2$.

(5) Determine the right-hand expression of each statement. Put all variables in the reference list and other variables or constants into an expression with operators such as +, -, *.

## 4 Testing UniPhier with JTT

JTT has been successfully applied in testing UniPhier. In this section, we report test experience on testing seven key optimizing modules in UniPhier. Among them, M1, M2, and M3 implement general-purpose optimizations, while M4, M5, M6, and M7 implement optimizations related to UniPhier architecture (the names of the modules are hidden because of business secret).

We specified test requirements, including the name of optimizations under test, data types, compiler directives, and so on, by using HLS in JTT. These HLS files were converted into EC++ programs automatically. Each EC++ program is compiled by UniPhier with an option to turn the optimization under test on and a reference compiler without any optimization turned on respectively. During compilation, statement coverage for UniPhier was recorded. If UniPhier failed the compilation but the reference compiler succeeded, a bug was detected. Then, the binaries were executed and the outputs were compared. If the outputs were not exactly the same, a bug was detected.

Before using JTT, UniPhier has been tested with benchmarks (SPEC CPU benchmark [14], EEMBC [5]) and public test suites (GCC test suite in version 3.4.6 and a test suite developed by MEI) for C/C++ compilers. Table 1 shows the statement coverage of the seven optimization modules by using benchmark and public test suite. From Table 1 we can see that, with JTT, statement coverage of the seven modules have all increased. Notably, the statement coverage of M4-M7 for benchmarks and public test suites are almost 0 because they are not aware any details about UniPhier architecture. The overall statement coverage of the seven modules has been improved from 37% to 72%.

| | M1 | M2 | M3 | M4 | M5 | M6 | M7 |
|---|---|---|---|---|---|---|---|
| Benchmark & Public Test Suite | 85% | 18% | 79% | 0% | 0% | 1% | 1% |
| With JTT | 88% | 54% | 80% | 63% | 49% | 68% | 56% |

**Table 1. Comparison of Statement Coverage of Optimizing Modules in UniPhier Without and With JTT**

UniPhier is also tested by more than 1,000 test programs written by test engineers in MEI. These test programs concern on modules M4-M7, and UniPhier-specific features in M1-M3. But because of the interleaving of the optimizations, data types, compiler directives and so on, these test programs are not enough for comprehensive testing. Another 6,000+ test programs produced by JTT are used to test UniPhier. These test programs cover most features of the seven modules in UniPhier. Six serious bugs have been found by such test programs produced by JTT. The number of all new bugs in the optimizing modules found by JTT accounts for 21% of the total number of bugs found during testing.

## 5 Concluding Remarks

In this paper, we report three major contributions for automated test program generation for an industrial optimizing compiler.

The first contribution is script-driven test program generation for a large system with complex input to meet test requirements. A test engineer can write test requirements in a simple script, and effective test programs are generated from these scripts. The script-driven test case generation can improve test efficiency.

The second contribution is model-based test program generation for optimizing modules. Model-based testing of real applications has attracted much interest. We also developed a model-based testing approach to produce desired programs to activate specific optimizations. The ability to generate specific test cases according to a model is a requirement for our testing.

The third contribution is the application of formal methods in a large complex system. Generally, modeling a large complex system, like an optimizing compiler, is very difficult. This makes application of formal methods in such a system impractical. In this paper, we do not model an entire optimizing compiler, but optimizations in this compiler, and gained much success as a consequence. This provides an evidence that the transition from entire systems to key components in the system not only makes the application of formal methods more easier and more practical, but also improves the quality of the entire system.

Overall, JTT can generate appropriate and adequate test programs for optimizing compilers with high quality and high efficiency. Our efforts in JTT provide a good example for testing a large complex system.

**Acknowledgements**

We wish to thank Shuichi Takayama, Akira Tanaka, Hajime Ogawa, Taketo Heishi, Shohei Michimoto of Matsushita Electric Industrial Co., Ltd.

Zhao Chen would like to thank Dr. David Lacy for his hosting in 2006 and for his discussion and comments on parallel optimization specification.

## References

[1] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[2] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, 8(4):343–353, July 1982.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[4] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Testing syntax and semantic coverage of java language compilers. *Information & Software Technology*, 41(1):15–28, 1999.

[5] The Embedded Microprocessor Benchmark Consortium, http://www.eembc.org/about/. *EEMBC*.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[7] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[8] P. A. Z. S. Kossatchev, A.S. and S. Zelenova. Application of model-based approach for automated testing of optimizing compilers. In *Proceedings of the International Workshop on Program Understanding*, pages 81–88, 2004.

[9] D. Lacey. *Program transformation using temporal logic specifications*. PhD thesis, Balliel College, University of Oxford, 2004.

[10] R. Lammel. Grammar testing. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 201–216, London, UK, 2001. Springer-Verlag.

[11] H. Li, M. Jin, C. Liu, and Z. Gao. Test criteria for context-free grammars. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 300–305, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Matsushita Electric Industrial Co., Ltd., http://www.semicon.panasonic.co.jp/e-micom/MicomFamily/uniphier/about.html. *What is UniPhier*, 2008.

[13] T. Rus and E. V. Wyk. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, 8(4):459–471, 1998.

[14] Standard Performance Evaluation Corporation, http://www.spec.org/cpu2006/. *SPEC CPU Benchmark*.

[15] Y. Xia. Loop optimization transformation using ctl specifications, in chinese. Master's thesis, Institute of Software, the Chinese Academy of Sciences, June 2006.