A Formal Approach to Parallelizing Compilers *

Teodor Rus † Eric Van Wyk †

Abstract

This paper describes parallelizing compilers which allow programmers to tune parallel program performance through an interactive dialog. Programmers specify language constructs that define sequential processes, such as assignment or for-loops, to be used as units of computation, while the compiler discovers the parallelism existent in the source program in terms of these units. Programmers may provide target machine architectural features used by compilers to coalesce sequential processes, controlling process granularity and ensuring process load balance.

1 Introduction

This paper describes two techniques used by an algebraic parallelizing compiler for program optimization. These techniques allow the programmer to control the granularity of the sequential processes executing a parallelized program and to find opportunities for parallelization and optimization between these processes. The programmer may specify *units of computation* describing the kind of computations, such as assignment statement, inner-loop-body, inner-loop, if-then-else, etc., that will be executed sequentially, as *units*, in the parallelized code. The programmer may additionally determine the minimum number of *units* to be used to create a sequential process. This allows the programmer to modify the granularity of the processes used in the dynamic scheduling of the program.

We also present in this paper a technique based on the formal methods of temporal logic and model checking to locate opportunities for optimization and parallelization in a program. Model checking is traditionally used to verify properties of concurrent and real-time systems. The properties to be verified are expressed by temporal logic formulas. A model checking algorithm determines the satisfiability of these formulas on the semantic model of the system represented as a labeled directed graph, called a *model* or Kripke structure [3]. Hence, by representing a source program as a model, and describing the conditions necessary for a particular optimization or parallelization as a temporal logic formula, the model checker can discover all locations in the program model that satisfy the formula and thus are candidates for the optimization. Thus, by providing a formal methodology to deal with program optimization, the compiler developer does not need to write code to discover optimization opportunities, only formulas that describe the required conditions. This simplifies the implementation of program optimizations and encourages more experimentation by the compiler developer with various optimizations thus improving the quality of the compiler. While we use this technique for parallelization, practically all other types of program analysis can be done with model checking.

*This work was partially supported by the grant NGT-51321 from the NASA Jet Propulsion Laboratory.

†Department of Computer Science, University of Iowa, Iowa City, IA

2 Process graph and model representations

The intermediate representations of the source program used by our parallelizing compiler are the *process graph* and the *process model*. The process graph is a directed graph in which nodes represent computations found in a source program and edges represent control and data dependency relationships between the computations. The process graph is not a traditional control flow graph whose edges direct the flow of control from one computation to the next. The nodes of a process graph represent sequential processes, i.e., stand alone computations, and the edges represent a minimal set of restrictions on the execution order of the computations required to ensure a correct execution of the computation represented by the graph.

To determine how a source program is broken into the pieces that are represented by the nodes of a process graph we define *units of computation* to be the types of computations that the programmer chooses to be executed sequentially. The constructs in the source program that are not units of computation are either too lightweight to stand as individual processes and must be combined with other program constructs, or have as their components units of computation and are therefore represented as *graphs* whose nodes are units of computation and edges are control and data dependencies between the nodes. Thus, while a unit of computation executes sequentially, it may perform in parallel with other units of computation in the process graph under the constraints imposed by the control and data dependency edges. For example, if *assignment statements* are units of computation, then an assignment statement in a source program will be represented as a node, whereas the expression on its right hand side is too lightweight to be a process; a sequence of assignment statements is represented as a collection of nodes each representing the individual assignment statements and edges representing the control and data dependencies between them. Since computations are represented by language constructs, the language specification rules defining valid language constructs allow us to formally define a unit of computation. That is, *a unit of computation is any valid construct recognized by one of the specification rules marked by the programmer as defining units of computation*. Hence, the programmer specifies which rules generate units of computation, and the compiler in turn generates sequential code from any construct recognized by these rules. By allowing the programmer to specify which constructs will be defined as units of computation the programmer can control the granularity of the processes executing the parallel program generated by the compiler. The processes represented by the nodes in the process graph are the *smallest* possible processes allowed by the programmer with respect to the specified units of computation. In addition, the programmer is allowed to modify the set of units of computation. Therefore, the compiler and programmer can engage in a dialog in which the programmer modifies the units of computation and the compiler displays the new process graph. This provides the programmer with immediate feedback about the structure of the parallel program and gives him or her the opportunity to adjust the process granularity.

A node in the process graph is represented by the tuple $<Types, State, Transition>$ where $Types$ is the set of data types of the variables and constants used in the computation, $State$ describes the variables, their types, and possible values, and $Transition$ describes the computation of the node as a transition system. We use $V_W(n)$ and $V_R(n)$ to denote the set of variables written and read by node n, respectively.

In the construction of graphs from nodes and other component graphs, we use two types of edges: those which define control dependency relationships and those which define data dependency relationships between units of computation. The simplest control dependency edges are labeled with $\prec$ and have as their source or target node one of the two special non-

computation nodes called *entry* (labeled e) and *exit* (labeled x) which mark the beginning and ending of a computation represented by a process graph.

A data dependency edge describes restrictions on the execution order of node computations caused when two computations access the same data. That is, a data dependency edge occurs between two nodes n_1 and n_2 if an instance of the computation at n_1 accesses the same variable or array element as an instance of n_2, at least one of these accesses is a write access, and the instance of n_1 executes before the instance of n_2 in a sequential execution of the program. A data dependency [1, 5] is classified as (1) a *data flow dependency*, and is denoted by $n_1 \xrightarrow{f} n_2$, if $V_W(n_1) \cap V_R(n_2) \neq \emptyset$, (2) a *data anti dependency*, and is denoted by $n_1 \xrightarrow{a} n_2$, if $V_R(n_1) \cap V_W(n_2) \neq \emptyset$, or (3) a *data output dependency*, and is denoted by $n_1 \xrightarrow{o} n_2$, if $V_W(n_1) \cap V_W(n_2) \neq \emptyset$. When arrays are accessed, the decision is more complicated than simple set intersection; a set of linear equations derived from the array index expressions must be solved [1]. The *distance* of a data dependency between two instances of computations n_1 and n_2 enclosed in loop ℓ is the number of iterations of loop ℓ between the instance of n_1 and the instance of n_2. If all instances of n_1 and n_2 that cause the data dependency have the same distance, the dependency is referred to as a constant distance data dependency, otherwise it is a non-constant distance data dependency. Data dependencies that have a distance of one or more are "carried" across loop iteration boundaries and are called loop carried data dependencies. When n_1 and n_2 are not in the same loop nest, the dependency is a loop crossing data dependency since it crosses from one loop nest into another.

In constructing graphs from nodes and component graphs, we restrict our presentation here to *functional*, *branch*, and *enumerated loop* compositions which correspond to sequential, if-then-else, and do-loop compositions in the source language. The simplest graph construction is the functional composition of two unit of computation nodes n_1 and n_2, which results in a graph of 4 nodes: e, x, n_1 and n_2, and control dependency edges: $e \xrightarrow{\prec} n_1$, $e \xrightarrow{\prec} n_2$, $n_1 \xrightarrow{\prec} x$, and $n_2 \xrightarrow{\prec} x$. We refer to this as functional composition since the computation computed by the graph is the functional composition of the two transition functions on nodes n_1 and n_2. If g_1 and g_2 are process graphs then $g_1 \oplus g_2$ is a the graph representing the functional composition of the computations represented by g_1 and g_2 and has the shape in Figure 1*(a)*.

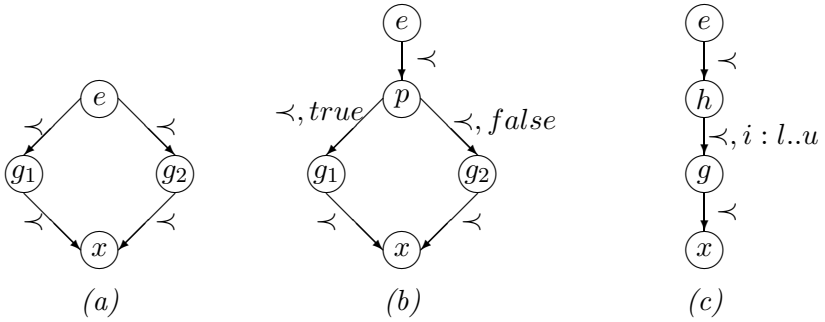

FIG. 1. *Functional (a), Branch (b), and Enumerated Loop (c) process graph compositions*

If there is a data dependency from a node n_1 of g_1 to a node n_2 of g_2 requiring the sequential execution of some component nodes, a data dependency edge is added, $n_1 \xrightarrow{d} n_2$, $d \in \{f, a, o\}$, which ensures the correct execution order of the computations at the nodes. Since there are no control dependency edges only the data dependency edges restrict the

execution order of the computations, thus the process graph exposes a high degree of parallelism The edges of the process graph represent the minimal set of restrictions on process execution order.

Branch composition of graphs g_1 and g_2 and predicate p has the shape in Figure 1*(b)*, where $true$ and $false$ are control dependencies.

An enumeration loop represents the repeated execution of a computation with different values associated with the loop index variable. The enumerated loop composition creates a graph in the shape in Figure 1*(c)* where $h \xrightarrow{i:l..u} g$ represents the simultaneous instantiation of *all* iterations of the loop body. Thus, we add data dependency edges to ensure correct execution order of the loop iterations. Note that the lack of control dependency edges between nodes of g is reminiscent of the lack of control dependency edges between nodes in a functional composition. In both cases they are not necessary since the data dependency edges which are added ensure a correct execution sequence of the computations.

Our process graphs are significantly different from most program flow graphs found in optimizing and parallelizing compilers [5]. First, nodes do not represent *basic blocks* - segments of target code that contain no branching statements. The nodes of a process graph represent source language constructs and their creation is controlled by the programmer by defining the set of *units of computation*. This allow the programmer to control the granularity of the sequential processes executing the parallel program. We also provide descriptive data dependency edges so that the control dependencies implicitly provided by the textual layout of the computation can be marked as irrelevant to the execution order of the statements in the program.

Process graphs are, however, very similar to the Kripke structures used in model checking. In fact, by representing some of the information labeling the nodes and edges in the process graph as atomic logical propositions, the projections of our graphs containing the nodes, edges, and propositions, which we call *process models*, are Kripke structures. Thus, many of the questions that optimizing and parallelizing compilers ask about programs can be represented as temporal logic formulas. These *questions* can then be *answered* by a model checking algorithm. The node propositions we use and their meaning are listed below:

ℓ_n	- unique label for node n	$unit$	- unit of computation nodes
e	- *entry* nodes	$branch$	- branch condition nodes
x	- *exit* nodes	$enum$	- enumerated loop header nodes

The edge propositions we use and their meaning are listed below:

$\prec$	- control dependency (dep.)	$D_{\ell,0}$	- constant distance data dep. with distance 0 for $enum$ loop node ℓ
$true$	- $true$ branch control dep.		
$false$	- $false$ branch control dep.	$D_{\ell,+}$	- constant distance data dep. with positive distance for $enum$ loop node ℓ
$enum$	- $enum$ control dep.		
f	- data flow dep.	$D_{\ell,?}$	- non constant distance data dep. for $enum$ loop ℓ
a	- data anti dep.		
o	- data output dep.	$D_{\ell,X}$	- data dep. that crosses loop boundary ℓ
V_`var`	- data dep. on variable `var`		

3 Temporal logic and model checking

Model checking is a formal verification technique which checks the correctness, according to some specification, of a system represented as a proposition labeled directed graph,

called a *model* or Kripke structure. Models used by our compiler are directed multigraphs whose nodes and edges are labeled with atomic propositions. Models have the form $\mathcal{M} = <N, E, P_n\colon AP_n \rightarrow 2^N, P_e\colon AP_e \rightarrow 2^E>$ where N is a set of finite nodes, E is a finite set of edges, AP_n maps node propositions to the set of nodes on which they hold, and AP_e maps edge propositions to the set of edges on which they hold. Since our models are multigraphs and thus may have multiple edges between the same two nodes, the source and target nodes of an edge may not uniquely identify the edge, thus we use the notation $source(e)$ and $target(e)$ to identify the source and target nodes respectively of an edge e. Also, a path $n_0, e_0, n_1, e_1, n_2, ...$ is the sequence of nodes n_i and edges e_i such that $\forall i \geq 0, n_i = source(e_i) \wedge n_{i+1} = target(e_i) \wedge e_i \in E$. Since the properties of a system are specified by temporal logic formulas written over the propositions which label the model $\mathcal{M}$, the process of model checking a temporal logic formula f over a model $\mathcal{M}$ determines on which nodes in the model the temporal formula holds. The temporal logic used in this paper is CTLe [4], an extension to CTL (Computational Tree Logic) developed by Clarke, Emerson and Sistla in [2]. CTLe formulas are recursively constructed from CTL formulas extended with CTLe edge formulas which quantify the paths examined by the temporal operators. This allows formulas to be written over the propositions on the *nodes* and *edges* of the model.

CTLe edge formulas are logical formulas over the edge propositions and are defined by the following rules:

1. $true$, $false$ and any atomic edge proposition $ap_e \in AP_e$ are CTLe edge formulas.
2. if f_1 and f_2 are CTLe edge formulas, so are $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$.

If an edge $e \in E$ satisfies an edge formula f for a model $\mathcal{M}$ we write $\mathcal{M}, e \models f$ or $e \models f$. Satisfaction of edge formulas is defined below:

$$\begin{array}{rcl} e \models ap_e & \text{iff} & e \in P_e(ap_e) \\ e \models \neg f & \text{iff} & not\ e \models f \\ e \models f_1 \wedge f_2 & \text{iff} & e \models f_1 \text{ and } e \models f_2 \\ e \models f_1 \vee f_2 & \text{iff} & e \models f_1 \text{ or } e \models f_2 \end{array}$$

CTLe formulas are similar to those used in [2] and are defined by the following rules:

1. $true$, $false$ and any atomic node proposition $ap_n \in AP_n$ are CTLe formulas.
2. if f_1 and f_2 are CTLe formulas, so are $\neg f_1$, $f_1 \vee f_2$, and $f_1 \wedge f_2$.
3. if f_1 and f_2 are CTLe formulas, and f_e is a CTLe edge formula, then $AX_{\{f_e\}} f_1$, $EX_{\{f_e\}} f_1$, $A[f_1 U_{\{f_e\}} f_2]$, and $E[f_1 U_{\{f_e\}} f_2]$ are also CTLe formulas.

The formula $AX_{\{f_e\}} f_1$ ($EX_{\{f_e\}} f_1$) is satisfied on a node if all (one or more) successors satisfy f_1 and the edges to these successors satisfy the edge formula f_e. The formula $A[f_1 U_{\{f_e\}} f_2]$ ($E[f_1 U_{\{f_e\}} f_2]$) is satisfied on a node if on all (one or more) paths beginning on this node there is a node on which f_2 holds, f_1 holds on all nodes before this node, and each intermediate edge in the path satisfies f_e.

The satisfaction of a CTLe formula f on a node $n \in N$ in a model $\mathcal{M}$ is denoted $\mathcal{M}, n \models f$ or $n \models f$. The satisfaction rules of these CTLe formulas are given by:

$$
\begin{array}{rll}
n \models ap_n & \text{iff} & n \in P_n(ap_n) \\
n \models \neg f & \text{iff} & \textit{not } n \models f \\
n \models f_1 \wedge f_2 & \text{iff} & n \models f_1 \text{ and } n \models f_2 \\
n \models f_1 \vee f_2 & \text{iff} & n \models f_1 \text{ or } n \models f_2 \\
n \models EX_{\{f_e\}} f_1 & \text{iff} & \exists e \in E,\ n = source(e) \wedge e \models f_e \wedge target(e) \models f_1 \\
n \models AX_{\{f_e\}} f_1 & \text{iff} & \forall e \in E,\ n = source(e) \Rightarrow (e \models f_e \wedge target(e) \models f_1) \\
n \models A[f_1\ U_{\{f_e\}}\ f_2] & \text{iff} & \forall\ paths\ (n_0, e_0, n_1, e_1, n_2, \ldots),\ n = n_0 \text{ and} \\
 & & \exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow (n_j \models f_1 \wedge e_j \models f_e)]] \\
n \models E[f_1\ U_{\{f_e\}}\ f_2] & \text{iff} & \exists\ a\ path\ (n_0, e_0, n_1, e_1, n_2, \ldots),\ n = n_0 \text{ and} \\
 & & \exists i[i \geq 0 \wedge n_i \models f_2 \wedge \forall j[0 \leq j < i \Rightarrow (n_j \models f_1 \wedge e_j \models f_e)]]
\end{array}
$$

To illustrate the model checking of CTLe formulas, we provide an example of a model in Figure 2 labeled with node propositions P and Q and edge propositions a, b, and c. Node 1 satisfies the formula $EX_{\{a \vee b\}}P$ since node 2 satisfies P and an edge from 1 to 2 satisfies $a \vee b$, but it does not satisfy $AX_{\{true\}}Q$ since although all edges will satisfy $true$, node 2 does not satisfy Q. Node 3 satisfies $A[Q\ U_{\{b\}}P]$ because on both paths (3,5,6) and (3,6) from node 3 Q holds until P holds, and b holds on all edges on those paths.

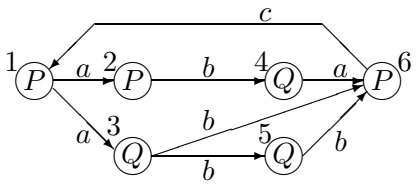

FIG. 2. *CTLe example*

4 Discovering optimization opportunities

During the compilation of the source program, the parallelizing compiler analyzes the process graph and process model of the source program looking for opportunities to optimize and parallelize the source program. We propose a technique for this analysis which eliminates the need for writing code. Instead, a compiler writer can write temporal logic formulas which describe the conditions required for a particular optimization or parallelization. This provides a concise, formal, and powerful way to describe optimization opportunities, it simplifies compiler writing by replacing the writing of code with the writing of formulas thus encouraging the compiler developer to experiment with more optimizations. In this section we present a simple code fragment and some CTLe formulas used to locate optimization opportunities in the process model.

One fundamental question a parallelizing compiler may ask is if the iterations of an enumerated loop are independent, implying that they can be executed concurrently. We consider this question on the simple loop in Figure 3. An enumerated loop has independent iterations if there are no loop carried data dependencies between the nodes representing the computations of the loop body. That is, an enumerated loop, with loop header node ℓ has independent iterations if all *unit* labeled direct and indirect successors of the loop header node reachable by a path whose edges are labeled by a control dependence $\prec$ do not have any successors reachable by a loop carried data dependency edge labeled with $D_{\ell,+}$ or $D_{\ell,?}$. These requirements can be stated as a CTLe formula which is tested for satisfaction on the

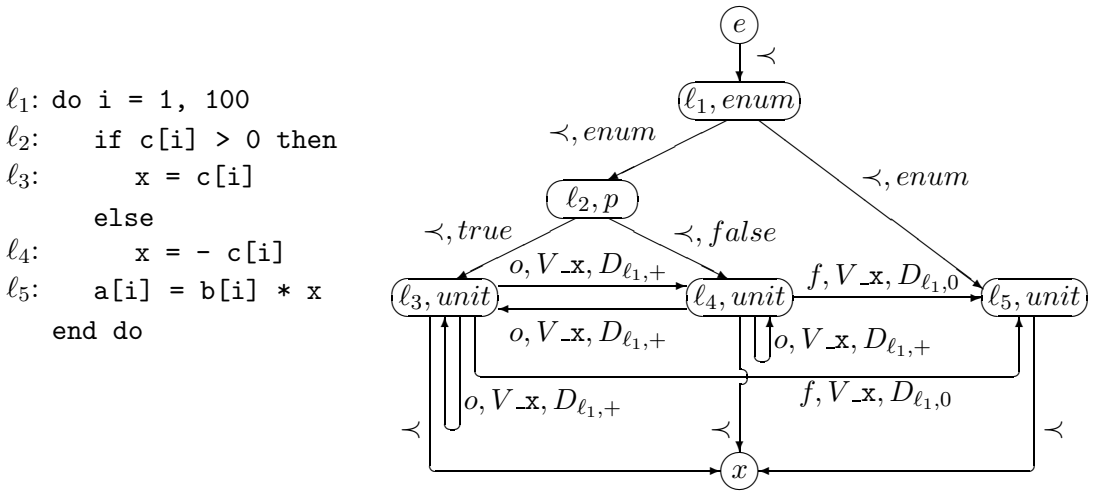

FIG. 3. *Source program and process model*

node ℓ_1 in the process model in Figure 3. That is, loop ℓ_1 has independent iterations if

$$\ell_1 \models enum \wedge A\ [\ true\ U_{\{\prec\}}\ (unit\ \wedge\ \neg EX_{\{D_{\ell_1,+}\ \vee\ D_{\ell_1,?}\}}\ true)\ ] \tag{1}$$

The model checker would report that the node ℓ_1 does not satisfy this CTLe formula. The data dependencies on scalar variable `x` prevent the parallelization of this loop as it now stands.

A common transformation which may enable loop parallelization is *scalar expansion* in which a scalar variable is *expanded* into an array such that the output of the program does not change but data dependencies on the scalar disappear. If the scalar, `x`, is written before it is read in all iterations of the enclosing loop ℓ_1, it can be expanded into an array indexed by the loop index variable, i.e. `x[i]`. We say that the variable `x` is *expanded over loop* ℓ_1. The semantics of the loop do not change, and the loop carried data dependencies disappear since each iteration of the loop works on its own array element which replaces the scalar. Scalar expansion over a loop ℓ is possible if there are no loop ℓ carried data *flow* dependencies on the scalar. (For simplicity, we assume the scalar is not used elsewhere in the code. This assumption can of course be removed by modifying the CTLe formula below.) The presence of a loop ℓ carried data flow dependency would indicate that the scalar is written in one iteration and read in a later iteration thus preventing the scalar expansion. We can again write these requirements as a CTLe formula. That is, the scalar `x` can be expanded over loop ℓ_1 if

$$\ell_1 \models A\ [\ true\ U_{\{\prec\}}\ (unit\ \wedge\ \neg EX_{\{f \wedge V\_x \wedge (D_{\ell_1,+} \vee D_{\ell_1,?})\}} true)\ ] \tag{2}$$

This formula is similar in structure to the previous iteration independence test and simply ensures that each *unit* node in the body of the loop ℓ does not have an ℓ loop carried data flow dependency on variable `x`. Node ℓ_1 satisfies this formula, thus the scalar variable could be expanded over the enumerated loop ℓ_1. After expanding `x` over the loop ℓ_1 there would be no loop carried dependencies in ℓ_1 and the loop would pass the iteration independence test in (1). Hence, the loop iterations could be executed concurrently.

Strictly speaking, a CTLe formula must be written over the propositions which occur in the model as was the case in the formulas above. However in the compiler, formulas

must be specified such that they will work for all source program models which will contain different propositions characterizing different constructs specified by the same specification rule. Therefore we specify temporal logic formulas as *formula macros* attached to the source language specification rules. During the compilation process when the compiler recognizes a construct specified by a rule r it expands the formula macros attached to r into CTLe formulas. This allows propositions appropriate to the program constructs or circumstance to be "plugged-into" the CTLe formulas using them as parameters of the formula macro that generates these formulas. For example, in testing for scalar expansion of the scalar whose name is referenced by the name `the_scalar` over the loop whose label is referenced by the name `the_loop`, we would obtain the appropriate formula as seen in (2) by plugging the appropriate propositions into the formula macro

$$A\ [\ true\ U_{\{\prec\}}\ (unit \wedge \neg EX_{\{f \wedge \$var(\texttt{the\_scalar}) \wedge (\$Dist+(\texttt{the\_loop}) \vee \$Dist?(\texttt{the\_loop}))\}} true)\ ]$$

where, in the example above, $\$var($`the_scalar`$)$ expands into the proposition $V\_\texttt{x}$, $\$Dist+$ (`the_loop`) expands into the proposition $D_{\{\ell_1,+\}}$, and $\$Dist?($`the_loop`$)$ expands into the proposition $D_{\{\ell_1,?\}}$. Thus, the formula macro can be expanded into the appropriate CTLe formula for any source program. While the formulas we have written here are certainly too restrictive in that they may reject loops for parallelization that could be parallelized, it is the process of writing formulas to detect optimization and parallelization opportunities that we want to emphasize. To find more parallelizable loops, we need only modify the CTLe formulas that detect them. We do not modify code to detect them.

5 Conclusions

We have presented a formal technique for program optimization by the compiler where process granularity is controlled by the programmer at the source language level. Programmers interact with the compiler specifying the computation contents of the nodes of the model. The optimizations are specified by the compiler designer using formula macros attached to the specification rules. The compiler constructs the process model of the program and expands formula macros into temporal formulas. A model checker verifies these temporal formulas checking for optimization opportunities. An experimental version of this compiler using Fortran-90 as the source language is under development.

References

[1] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic, Boston, 1988.

[2] E. Clarke, E. Emerson, and A. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems, 8 (1986), pp. 244–263.

[3] S. Kripke, *Semantical analysis of modal logic i: Normal modal propositional calculi*, Zeitschrift f. Math. Logik und Grundlagen d. Math., 9 (1963).

[4] T. Rus and E. Van Wyk, *Model checking tools for parallelizing compilers*, in Second International Workshop on Formal Methods for Parallel Programming: Theory and Applications, Proceedings, April 1 1997. Paper available at http://www.cs.uiowa.edu/~rus.

[5] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, California, 1996.