Static Analysis for Guarded Code

Ping Hu

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

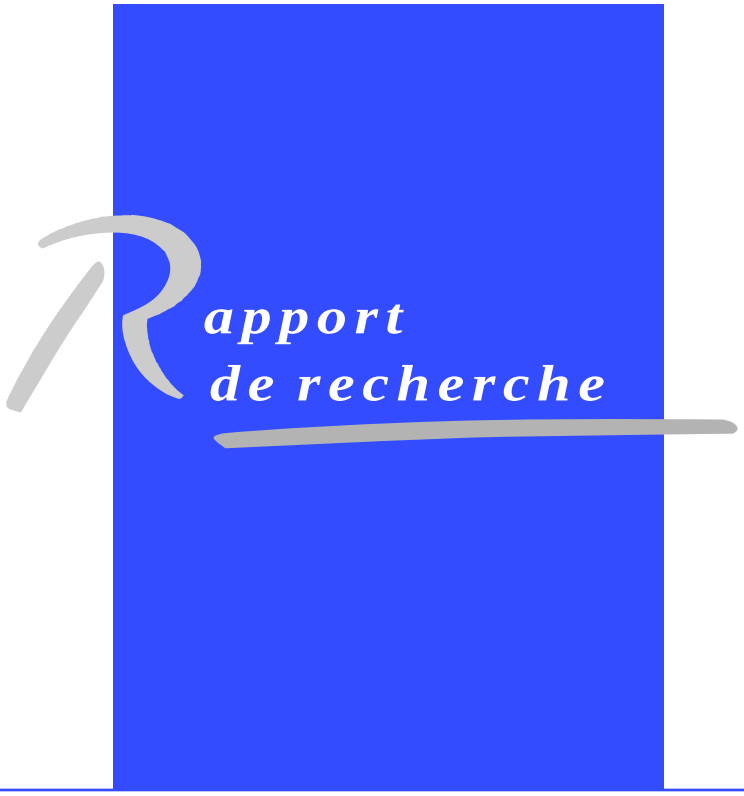
Static Analysis for Guarded Code

Ping Hu

N ° 3979

25 Juillet 2000

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--3979--FR+ENG

Static Analysis for Guarded Code

Ping Hu

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n ° 3979 — 25 Juillet 2000 — 19 pages

Abstract: *Guarded(predicated) execution*, as a new hardware feature, has been introduced into today's high performance processors. *Guarded execution* can significantly improve the performance of programs with conditional branches, and meanwhile also poses new challenges for conventional program analysis techniques.

In this paper, we propose a static semantics inference mechanism to capture the semantics information of guards in the context of guarded code. Based on the semantics information, we extend the conventional definitions regarding program analysis in guarded code, and develop the related guard-aware analysis techniques. These analyses include control flow analysis, data dependence analysis and data flow analysis as well.

Key-words: Guarded(predicated) execution, Static analysis, Control flow, Data-dependence, Data-flow

(Résumé : tsvp)

This research was partially supported by the ESPRIT IV reactive LTR project OCEANS, under contract No. 22729.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Analyse Statique de Code Gardé

Résumé : L'exécution gardée est de plus en plus souvent introduite dans les dispositifs matériels des nouveaux processeurs à haute performance. L'exécution gardée peut améliorer de manière significative la performance d'un programme avec branchements conditionnels. Cependant, elle pose également de nouveaux problèmes pour les techniques conventionnelles de compilation.

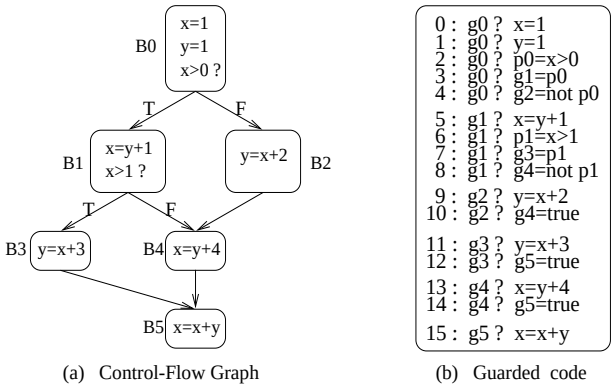
Dans cet article, nous proposons un mécanisme d'inférence statique de la sémantique pour saisir l'information de la sémantique des gardes dans le contexte du code gardé. Sur la base de l'information sémantique, nous étendons les définitions conventionnelles concernant l'analyse de programmes au code gardé, et développons les techniques d'analyse associées *en tenant compte des gardes*. Ces analyses incluent l'analyse de flots de contrôle, l'analyse de dépendance de données ainsi que l'analyse de flots de données.

Mots-clé : Exécution gardée, Analyse statique, Flot de contrôle, Dépendance de données, Flot de données

1 Introduction

High performance compilation techniques rely heavily on effective program analysis. Sufficient and precise information on a program is critical to program optimization as well as to program parallelization.

Guarded(predicated) execution [10, 9], as a new hardware feature, has been introduced into more and more high performance processors. This hardware feature provides an additional boolean register for each operation to guard whether the operation will be executed or not. If the value of the register is *true*, then the operation will be executed normally, otherwise the operation will be collapsed after initiating the execution of the operation. Such a register in an operation is termed the *guard* of the operation and the operation is said to be a *guarded(predicated) operation*. To support *guarded execution*, a compiler algorithm, called *if-conversion* [2, 9, 3, 6], converts programs with conditional branches into guarded code. As a result, *if-conversion* removes conditional branches from programs. Figure 1 shows an example of guarded code, which has been if-converted from the control flow graph given on the left of the figure.



(a) Control-Flow Graph (b) Guarded code

Figure 1: An example of guarded code

Guarded execution can significantly improve the performance of a program with conditional branches due to two main facts. First, *if-conversion* enlarges the size of basic blocks and thereby provides a large number of opportunities to extract the available parallelism from the enlarging scheduling scope. Second,

the elimination of branches can avoid high branch misprediction penalties so as to improve branch dynamic behavior. However, the introduction of *guarded execution* also proposes new challenges for conventional program analysis techniques when applied to guarded code.

For instance, for two successive guarded operations below,

```
g1? x = y + 1
g2? x = x * 2
```

does there exist any data-dependence between them? Will the value of variable x defined in the first operation be redefined by the second operation? If we ignore the effect of guards on the operations, the answer should be 'yes' to both questions. However, if the two guards $g1$ and $g2$ are disjoint(i.e. they never evaluate to *true* at same time), there is indeed no data-dependence between them. Only when $g2$ is always *true* as long as $g1$ is *true*, will the variable x in the first operation be redefined(killed) by the second operation and is not reachable(alive) after the second one. Hence, the logical relations among guards have to be taken into consideration in the analysis techniques.

The authors in [4] have suggested P-facts to extract and represent the disjointedness relations between guards. These P-facts are used to analyze live-variable ranges for register allocation. But the extraction mechanism for disjointedness relations of guards is very sensitive to the instruction set architecture since it depends upon the instruction scheme of the HPL PlayDoh architecture [8]. Another data-structure for tracking guard relations, proposed in [7], is the *predicate partition graph*. This graph-based data-structure is used to provide query information for data-flow analysis and register allocation in [5]. Similar to [4], the construction of the partition graph is based upon the HPL PlayDoh instruction set. A rather different approach developed in [12] is to apply *reverse if-conversion* to convert guarded code back to an explicit conditional branch structure where traditional analysis techniques can be applicable. In contrast, we expect to develop analysis techniques that would be directly applicable to guarded code.

In this paper, we propose a static semantics inference mechanism in the context of guarded code, which can capture the semantics information of guards directly from guarded code, and allows us to analyze the logical relations among guards.

Based on the semantics information, we extend the conventional definitions regarding program analysis in the context of guarded code, and develop the related guard-aware analysis techniques. These guard-aware analyses include not only data-flow analysis but also control-flow and data-dependence analysis. The guard-aware control-flow analysis enables us to achieve the traditional results of control-flow analysis, such as dominance, post-dominance and control-equivalence, etc. The control-equivalence analysis has been used to reduce the number of guards required in guarded code. The guard-aware data-dependence analysis can avoid a lot of dependences so as to provide more opportunities for exploiting and extracting parallelism in guarded code.

The remainder of the paper is organized as follows. Section 2 presents the semantics inference mechanism. Section 3 presents the guard-aware control-flow analysis. The guard-aware data-dependence and data-flow analysis will be presented in Sect. 4 and Sect. 5, respectively. The last section gives the concluding remarks and outlines our future work.

2 Semantics Analysis for Guards

The semantics of a guard is a logical proposition which consists of predicate variables(i.e. branch conditions, e.g. p_1 and p_2 in the example of Fig. 1) and three basic logical operators($\wedge, \vee$ and $\neg$). That implies an operation will be executed only when its guard's semantics is $true$, i.e. the proposition evaluates to $true$.

A judgment $C \vdash S$ is introduced to denote that, from C, a given segment of guarded code, one can deduce S, a set of semantics of all guards in the guarded code. We have defined three inference rules for the reduction of guard semantics in Tab. 1.

Rule *taut* identifies that g_0 is a true-guard whose value is always true. Rule $fork$ describes, if the semantics of guard g_1 is known as l_1 in S, after the execution of the guarded operation $(g_1?\ g_2 = l_2)$, the semantics of guard g_2 is the proposition $dnf^1(l_1 \wedge l_2)$, i.e. the conjunction of the g_1's semantics l_1 and the condition l_2 under which the operations guarded by g_2 will be executed. The difference of the third rule $join$ from the second rule $fork$ is that guard

[1]Function dnf returns a logical proposition in disjunction normal form.

Table 1: Guard semantics analysis

$$\vdash \{g_0 = true\} \qquad \text{(taut)}$$

$$\frac{C \vdash S \cup \{g_1 = l_1\}}{C; (g_1?\ g_2 = l_2) \vdash S \cup \{g_1 = l_1\} \cup \{g_2 = dnf(l_1 \wedge l_2)\}} \quad \text{(fork)}$$

$$\frac{C \vdash S \cup \{g_1 = l_1\} \cup \{g_2 = l_2\}}{C; (g_1?\ g_2 = l_3) \vdash S \cup \{g_1 = l_1\} \cup \{g_2 = dnf((l_1 \wedge l_3) \vee l_2)\}} \quad \text{(join)}$$

g_2 has already had a semantics definition l_2 in S. Hence, the new semantics of g_2 after the guard operation $(g_1?\ g_2 = l_3)$ should be $dnf((l_1 \wedge l_3) \vee l_2)$, i.e. the disjunction of the current g_2's semantics $(l_1 \wedge l_3)$ and the previous g_2's semantics l_2.

These three inference rules are applied to deduce the guard semantics for the above example in Fig. 1. The detail for the inference procedure is demonstrated in Tab. 2.

The guarded operations in the first column have been analyzed one by one via the use of the inference rules. The names of the rules applied to the operations are shown in the third column. The details of how to apply the rules to the operations are given in the second column. For instance, the only applicable rule is *taut* at the beginning of all the operations, as shown in the first line. For the operation $(g_0?\ g_1 = p_0)$, we can apply rule $fork$ to obtain g_1's semantics $dnf(true \wedge p_0)$, i.e. p_0. For the operation $(g_2?\ g_4 = true)$, g_4 has a semantic definition $(p_0 \wedge \neg p_1)$ in S at this moment, we thus apply rule $join$ and obtain its new semantics $dnf((\neg p_0 \wedge true) \vee (p_0 \wedge \neg p_1))$, i.e. $(\neg p_0 \vee \neg p_1)$.

The final semantics set for all the guards is achieved as follows,

$$\{g_0 = true,\ g_1 = p_0,\ g_2 = \neg p_0,\ g_3 = p_0 \wedge p_1, g_4 = \neg p_0 \vee \neg p_1,\ g_5 = true\}$$

A function Sem is employed to return the semantics of a guard. For example, $Sem(g_0) = true$, $Sem(g_1) = p_0$ and $Sem(g_2) = \neg p_0$, etc.

Table 2: The deduction for the semantics of the guards

C	S	Rule
	$g_0 = true$	taut
$g_0?\ x = 1$ $g_0?\ y = 1$ $g_0?\ p_0 = x > 0$		
$g_0?\ g_1 = p_0$	$g_1 = dnf(true \wedge p_0) = p_0$	fork
$g_0?\ g_2 = \neg p_0$	$g_2 = dnf(true \wedge \neg p_0) = \neg p_0$	fork
$g_1?\ x = y + 1$ $g_1?\ p_1 = x > 1$		
$g_1?\ g_3 = p_1$	$g_3 = dnf(p_0 \wedge p_1) = p_0 \wedge p_1$	fork
$g_1?\ g_4 = \neg p_1$	$g_4 = dnf(p_0 \wedge \neg p_1) = p_0 \wedge \neg p_1$	fork
$g_2?\ y = x + 2$		
$g_2?\ g_4 = true$	$g_4 = dnf((\neg p_0 \wedge true) \vee$ $(p_0 \wedge \neg p_1)) = \neg p_0 \vee \neg p_1$	join
$g_3?\ y = x + 3$		
$g_3?\ g_5 = true$	$g_5 = dnf((p_0 \wedge p_1) \wedge true)$ $= p_0 \wedge p_1$	fork
$g_4?\ x = y + 4$		
$g_4?\ g_5 = true$	$g_5 = dnf(((\neg p_0 \vee \neg p_1) \wedge true)$ $\vee (p_0 \wedge p_1)) = true$	join
$g_5?\ x = x + y$		

An immediate application of the semantics set is dead-code elimination in the context of guarded code. We can eliminate those operations whose guards are *false* in the semantics set, because it is clear that this kind of operations will never be executed in any cases.

The semantics sets of guards also provides a good foundation for the analysis of guarded code. They enable us to analyze the logical relations between guards, and to develop the guard-aware analysis techniques. These analyses include control-flow, data-dependence and data-flow analysis, which are presented respectively in the following sections.

3 Guard-Aware Control Flow Analysis

Although a lot of information about control flow paths has been lost in if-converted code, we can still achieve some conventional results of control flow analysis, such as dominance, post-dominance and control-equivalence, etc, with the support of the semantics sets of guards obtained in the previous section.

3.1 Dominance analysis

Dominance is a fundamental concept in control flow analysis. A node m of a flow graph *dominates* node n if every path from the initial node of the flow graph to n goes through m, see [1]. This definition implies that if node n is visited from the initial node along an arbitrary path, then m must have been visited along the path as well. We extend the definition to guarded operations,

Definition of *Dominator*

A guarded operation $op1$ *dominates* guarded operation $op2$ if each time $op2$ is executed, then $op1$ has definitely been executed.

$$Dom(op1, op2) =_{df} \\ (op1 \preccurlyeq op2) \wedge Taut(Guard(op2) \rightarrow Guard(op1))$$

The definition of *Dominator* is represented by a boolean function $Dom(op1, op2)$, which returns $true$ if $op1$ *dominates* $op2$, otherwise returns $false$. The symbol $\preccurlyeq$ denotes the execution initiation order of operations, for instance, $op1 \preccurlyeq op2$

represents that an execution initiation for $op1$ is not later than that for $op2$. The boolean function $Taut$ verifies whether a logical proposition is a tautology(i.e. the proposition always evaluates to true),

$$Taut(p) =_{df} \begin{cases} true & \text{proposition } p \text{ is a tautology} \\ false & \text{otherwise} \end{cases}$$

It is a decidable problem to check whether a given logical proposition is a tautology or not. Its computational complexity is $O(2^n)$ in the worst case, where n is the number of predicate variables in the proposition.

The function *Guard* returns the semantics of the guard of an operation from the semantic sets of guards, which are obtained after the semantics analysis of guards presented in Sect. 2. The symbol $\rightarrow$ is used to denote the logical implication operator.

For the above given example, we can verify that the first operation $op0$ *dominates* all the operations. For instance, $op0$ *dominates* $op5(g_1?\ x = y + 1)$ because

$$\begin{array}{rl} & Taut(Guard(op5) \rightarrow Guard(op0)) \\ \Rightarrow & Taut(Sem(g_1) \rightarrow Sem(g_0)) \\ \Rightarrow & Taut(p_0 \rightarrow true) \Rightarrow true \end{array}$$

It can be further verified that $op5$ *dominates* all the operations guarded by g_3 because

$$\begin{array}{rl} & Taut(Sem(g_3) \rightarrow Sem(g_1)) \\ \Rightarrow & Taut((p_0 \wedge p_1) \rightarrow p_0) \Rightarrow true \end{array}$$

But $op5$ does not dominate the operations guarded by g_4 because

$$\begin{array}{rl} & Taut(Sem(g_4) \rightarrow Sem(g_1)) \\ \Rightarrow & Taut((\neg p_0 \vee \neg p_1) \rightarrow p_0) \Rightarrow false \end{array}$$

The dual *dominator* notion is *post-dominator*. A node n of a flow graph *post-dominates* node m if every path from m to any exit of the flow graph goes through n, see [1]. The definition of *post-dominator* is extended to guarded code in the same manner as *dominator*.

Definition of *Post-dominator*

A guarded operation $op2$ *post-dominates* guarded operation $op1$ if each time

$op1$ is executed, then $op2$ will definitely be executed.

$$Pdom(op2, op1) =_{df} \\ (op1 \preccurlyeq op2) \wedge Taut(Guard(op1) \rightarrow Guard(op2))$$

Let us have a look again at the example. It can be seen the last operation $op15$ *post-dominates* all the operations. In addition, the operations guarded by g_4 *post-dominate* those guarded by g_2 because

$$\begin{array}{rl} & Taut(Sem(g_2) \rightarrow Sem(g_4)) \\ \Rightarrow & Taut(\neg p_0 \rightarrow (\neg p_0 \vee \neg p_1)) \Rightarrow true \end{array}$$

As a corollary, it is trivial to verify that any guarded operation *dominates* and *post-dominates* itself.

3.2 Control-equivalence

In general, we are interested in two kinds of control-flow relationships in the analysis of control-flow, control-dependence and control-equivalence. Most control-dependences have been converted into data-dependences in if-converted code. The analysis of data-dependence will be presented in the next section. Here, we focus on the analysis of control-equivalence.

Definition of *Control-equivalence*

An operation $op1$ is *control-equivalent* to operation $op2$ iff

1. $op1$ *dominates*(or *postdominates*) $op2$
2. $op2$ *postdominates*(or *dominates*) $op1$

Analysis of *Control-equivalence*

A boolean function $ConEq$ is used to verify whether two guarded operations are *control-equivalent*.

$$ConEq(op1, op2) =_{df} (Dom(op1, op2) \wedge Pdom(op2, op1)) \\ \vee (Dom(op2, op1) \wedge Pdom(op1, op2))$$

According to the above definitions of Dom and $Pdom$, the function $ConEq$ can be simplified as

$$ConEq(op1, op2) =_{df} Taut(Guard(op1) \leftrightarrow Guard(op2))$$

Moreover, a guard g_1 is *control-equivalent* to guard g_2 iff $Sem(g_1) \leftrightarrow Sem(g_2)$ is a tautology, i.e.

$$ConEq(g_1, g_2) =_{df} Taut(Sem(g_1) \leftrightarrow Sem(g_2))$$

In the above example, g_0 is *control-equivalent* to g_5 as $(true \leftrightarrow true)$ is a tautology. The operations guarded by g_0 are thus *control-equivalent* to those guarded by g_5.

All guards that are mutually *control-equivalent* form a *control-equivalence* class. Because the guards in a *control-equivalence* class are *control-equivalent*, they can share the same name so as to reduce the number of required guards. Therefore, the guarded code in the example can be improved by renaming g_5 to the control-equivalent guard g_0, and eliminating all the operations for the assignment of g_5. The improved guarded code as well as the original code are shown in Fig. 2.

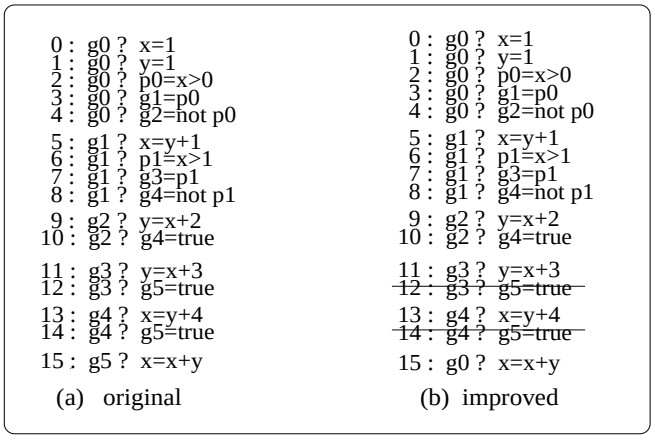


Figure 2: The improved guarded code

Remark: The functions Dom, $Pdom$ and $ConEq$ are applicable only to the operations in the same loop iteration. More precisely, given two operations

$op1$ and $op2$ from a loop body, if $Dom(op1, op2)$ evaluates to $true$, that just means $op1$ dominates $op2$ in the same iteration, and does not mean $op1$ from an iteration dominates $op2$ from another different iteration. This is not surprising because the traditional dominance relationship based on a control flow graph is also limited in the same iteration. It is possible that there is no dominance between two nodes from different iterations, even though conventional dominance analysis determines that one node dominates the other.

4 Guard-Aware Data Dependence Analysis

A data-dependence holds between two operations when one of them computes values needed by the other. Data-dependences directly determine the available parallelism in a program since they decide the execution order of operations. To guarantee the semantic correctness of a program, scheduling the operations for extracting parallelism must honor data-dependences.

Traditionally, data-dependences are divided into three classes,

1. *Flow-dependence*
   An operation *flow-depends* on another operation if a variable used in the former is defined by the latter.

2. *Anti-dependence*
   An operation *anti-depends* on another operation if a variable defined in the former is used by the latter.

3. *Output-dependence*
   An operation *output-depends* on another operation if the former defines the same variables as the latter.

In fact, as Wolfe mentions in [13], there does not exist a data-dependence between two operations respectively from *then* and *else* edges of a conditional branch since they will never be executed at the same time. We further extend this fact as:

> *there exists no data-dependence between two operations that are never executed along the same execution path.*

If two guarded operations are not executed along the same execution path, then the two guards of both operations don't evaluate to $true$ at same time. Such operations are said to be disjoint. We use a boolean function $Disjoint$ to determine whether two guarded operations are disjoint, i.e.

$$Disjoint(op1, op2) =_{df} Taut(\neg(Guard(op1) \wedge Guard(op2)))$$

We formally extend the above data-dependence definitions in the guard-aware data-dependence analysis.

1. **flow-dependence:** $Dflow(op1, op2) =_{df}$
   $(op2 \prec op1) \wedge (Use(op1) \cap Def(op2) \neq \phi) \wedge \neg Disjoint(op1, op2)$

2. **Anti-dependence:** $Danti(op1, op2) =_{df}$
   $(op2 \prec op1) \wedge (Def(op1) \cap Use(op2) \neq \phi) \wedge \neg Disjoint(op1, op2)$

3. **Output-dependence:** $Doutput(op1, op2) =_{df}$
   $(op2 \prec op1) \wedge (Def(op1) \cap Def(op2) \neq \phi) \wedge \neg Disjoint(op1, op2)$

where $(op2 \prec op1)$ represents that an execution initiation for $op2$ is earlier than that for $op1$, as mentioned above. The function Use returns the set of all variables used(read) in an operation and the function Def returns the set of all variables defined(written) in an operation.

Return to the example. The operation $op9(g_2?\ y = x + 2)$ would have depended on $op5(g_1?\ x = y + 1)$ if we had not taken take into account the effects of the guards. However,

$$\begin{array}{ll} & Disjoint(op5, op9) \\ \Rightarrow & Taut(\neg(Guard(op5) \wedge Guard(op9))) \\ \Rightarrow & Taut(\neg(Sem(g_1) \wedge Sem(g_2))) \\ \Rightarrow & Taut(\neg(p_0 \wedge \neg p_0)) \Rightarrow true \end{array}$$

$op5$ and $op9$ are disjoint, and thus $op9$ is not data-dependent on $op5$. Moreover, we can deduce that g_2 and g_3 are also disjoint, and thus the operations guarded

by g_3 do not depend on those guarded by g_2. For the same reason, there is no data-dependence between the operations guarded by g_3 and those guarded by g_4.

From this example, we can see the guard-aware analysis for data-dependences has effectively got rid of a large number of data-dependences between the disjoint operations. That will provide more opportunities for exploiting and extracting parallelism in guarded code.

Remark: It is not possible to statically determine two operations from different loop iterations are disjoint, even though they are known to be disjoint in the same iteration. Hence, when the function $Disjoint$ is applied to the operations from different iterations, its value has to be conservatively supposed to be $false$ in order to guarantee the correction of the static guard-aware data-dependence analysis.

5 Guard-Aware Data-Flow Analysis

5.1 Reaching definition

The notion of *reaching definition* concerns whether a definition in an operation can reach some point of a program. Conventionally, a value of a variable defined in an operation $op1$ can reach another operation $op2$ if this variable is not redefined along an execution path from $op1$ to $op2$, refer an example in [11].

We represent the *reaching definition* in the presence of guarded code by a boolean function $Reach$.

$$Reach(op1, op2) =_{df}$$
$$(op1 \prec op2) \wedge (Def(op1) \neq \phi) \wedge \neg Kill(op1, op2)$$

where the additional condition $(Def(op1) \neq \phi)$ is used to guarantee that there is a variable definition in the operation. The boolean function $Kill$ represents whether or not a variable definition in $op1$ would be killed by some operation between $op1$ and $op2$. A variable definition in $op1$ is killed before reaching $op2$

> *when the guard of op1 evaluates to true, there is some operation between op1 and op2 that would redefine this variable, and its guard always evaluates to true.*

Suppose $\bigvee \phi = false$,

$$Kill(op1, op2) =_{df} Taut(Guard(op1) \rightarrow \bigvee\{ Guard(op) \mid (op1 \prec op \prec op2) \wedge (Def(op) \cap Def(op1) \neq \phi)\})$$

A similar *reaching definition* proposed in [4] is that the variable definitions in a guarded operation can reach some point in a guarded code when the guard of the operation evaluates to $true$, and meanwhile all the guards of the operations that would redefine the variable evaluate to $false$. For the example,

```
0 : g0? x = 1
......
5 : g1? x = y + 1
......
9 : g2? y = x + 2
```

the value of variable x defined in $op0$ can reach $op9$ only when g_0 evaluates to $true$, and g_1 evaluates to $false$. Here, the question is how to determine statically that the value of $(g_0 \wedge \neg g_1)$, i.e. $(true \wedge \neg p_0)$, is $true$ or $false$. In fact, this is an undecidable problem at compile time even if we had the semantics set for the guards.

Compared with our *reaching definition*, the value of x in $op0$ can reach $op9$ if g_1 does not always evaluate to $true$ while g_0 evaluates to $true$. This implies that there exists a path(when g_1 is $false$) so that this value of x can flow through $op5$ and reach $op9$. This obviously agrees with the original *reaching definition*. Conversely, if g_1 always evaluates to $true$ while g_0 evaluates to $true$, this implies that $op5$ post-dominates $op0$. In this case, the value of x in $op0$ will definitely be killed by $op5$ in any case, and therefore can not reach $op9$. Moreover, it is a decidable problem to check whether a logical proposition is a tautology or not, as mentioned above.

Here, we have

$$\begin{array}{ll} & Kill(op0, op9) \\ \Rightarrow & Taut(Sem(g_0) \rightarrow Sem(g_1)) \\ \Rightarrow & Taut(true \rightarrow p_0) \Rightarrow false \end{array}$$

the value of x in $op0$ thus can reach $op9$. In contrast, this value can not reach the last operation $op15$ since

$$\begin{array}{ll} & Kill(op0, op15) \\ \Rightarrow & Taut(Sem(g_0) \rightarrow \bigvee\{Sem(g_1), Sem(g_4)\}) \\ \Rightarrow & Taut(true \rightarrow \bigvee\{p_0,\ (\neg p_0 \vee \neg p_1)\}) \Rightarrow true \end{array}$$

In the next subsection, we utilize the two functions $Reach$ and $Kill$ to define the guard-aware data-flow equations.

5.2 Guard-aware data flow equations

Before giving the guard-aware data-flow equations, we recall the conventional equation for data-flow analysis. The information(for instance, variable definitions) reaching the end of a basic block is that information which is either generated within the block or enters the beginning but is not killed by the block. The formal expression of this statement is the following well-known data-flow equation [1],

$$Out(B) =_{df} Gen(B) \cup (In(B) - Kill(B)) \quad where$$

- $Out(B)$: the set of all operations whose variable definitions can reach the end of B;

- $Gen(B)$: the set of the operations in B whose variable definitions can reach the end of B;

- $In(B)$: the set of all operations whose variable definitions enter at the beginning of B;

- $Kill^2(B)$: the set of the operations in $In(B)$ whose variable definitions are killed by B.

[2]For the sake of avoiding too many notations, the function names can be overloaded here.

In the context of guarded code, a basic block should be a maximal set of consecutive guarded operations with one entry point and one exit point, which is a so-called guarded block. Given such a guarded block GB, we can obtain the variable definitions reaching the end of the guarded block via the analysis of the *reaching definition* presented in the previous subsection. Suppose *end* is a virtual empty operation to represent the end point of GB, *end* is thus the successor of the last operation in GB. Table 3 gives the guard-aware data-flow equations.

Table 3: Guard-aware data-flow equations

$$Out(GB) =_{df} Gen(GB) \cup (In(GB) - Kill(GB)) \quad \text{where}$$

$$Gen(GB) =_{df} \{\ op_i \mid (op_i \in GB) \wedge Reach(op_i, end)\ \}$$

$$In(GB) =_{df} \bigcup_{P \in pred(GB)} Out(P)$$

$$Kill(GB) =_{df} \{\ op_i \mid (op_i \in In(GB)) \wedge Kill(op_i, end)\ \}$$

The guarded code in the example of Fig. 1 forms a guarded block GB. Its $Gen(GB)$ is

$$\{1,\ 2,\ 3,\ 4,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11\}$$

since the definitions of variable x in operations 0,5,13 are killed before reaching the end of the block. The final $Out(GB)$ is same as the $Gen(GB)$ because its $In(GB)$ is empty.

An important application of data-flow equations is to analyze live-variable ranges for register allocation. We believe the above guard-aware data-flow analysis would form an essential base for developing guard-aware register allocation techniques.

6 Conclusion

In this paper, we have presented a static semantics inference mechanism to capture the semantics information on guards, which provides a unified framework to develop the guard-aware analysis techniques, such as control flow analysis, data dependence analysis and data flow analysis. These guard-aware analyses provide the essential information to support optimizing and parallelizing compilation techniques on processors with guarded execution.

Guarded execution actually provides more opportunities and greater flexibility for program optimization. In future work, we intend to develop the optimization techniques especially for guarded code.

Acknowledgements

The author would like to thank Christine Eisenbeis and François Thomasset for their insightful comments and suggestions which improved the quality of this paper. The author would also like to thank Gang Chen and Michael Liebelt for their support in the work. Special thanks to Richard James for the correction of English faults.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] J.R. Allen, K. Kennedy, C. Portefied, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[3] J.C. Dehnert and R.A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1/2):181–227, May 1993.

[4] A.E. Eichenberger and E.S. Davidson. Register allocation for predicated code. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 180–191, November 1995.

[5] D.M. Gillies, D.R. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 114–125, December 1996.

[6] J. Hoogerbrugge and L. Augusteijn. Instruction Scheduling for TriMedia. *Journal of Instruction-Level Parallelism*, 1, February 1999.

[7] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 100–113, December 1996.

[8] V. Kathail, M.S. Schlansker, and B.R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett Packard Laboratories, February 1994.

[9] J.C.H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett Packard Software and Systems Laboratory, May 1991.

[10] B.R. Rau, W.L. Yen, and R.A. Towle. The cydra 5 departmental supercomputer. *IEEE Computer*, pages 12–35, January 1989.

[11] F. Thomasset. Analyse de flots de données : introduction. *Notes de cours : DEA d'Informatique*, March 1999. http://www-rocq.inria.fr/~thomasse/DEA/.

[12] N.J. Warter, S.A. Mahlke, W.W. Hwu, and B.R. Rau. Reverse if-conversion. In *Proceedings SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 290–299, June 1993.

[13] M. Wolfe. *High performance compilers for parallel computing.* Addison Wesley, 1996.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399