Efficient Annotation Inference for an Extended Static Checker

Cormac Flanagan
Compaq SRC
cormac.flanagan@compaq.com

K. Rustan M. Leino
Compaq SRC
rustan.leino@compaq.com

Michael Y. Levin
University of Pennsylvania
milevin@cis.upenn.edu

Abstract

(Submission to SAS'01.) A modular static program checker relies on annotations specifying module interfaces. Writing annotations is a burden to the programmer. The Houdini algorithm is a whole-program analysis that reduces this burden by inferring many annotations automatically. The basic Houdini algorithm infers useful annotations for an extended static checker, but is very slow.

This paper describes an optimized, distributed version of the Houdini algorithm that yields an order of magnitude improvement in performance. The paper gives a formal description of the inference problem and of the optimized algorithm, and describes the implementation and evaluation of the new algorithm.

1 Introduction

The goal of static program checking is to detect programming errors early in the development cycle, when they are significantly easier to fix. A common example of a static checker is a type checker, which detects errors such as the application of a function to inappropriate argument values. Another static checker is the Compaq Extended Static Checker for Java (ESC/Java), which checks for additional errors that are not caught by traditional type systems, such as dereferencing the null pointer, indexing an array outside its bounds, or accessing a shared variable without holding its protecting lock [Ext]. ESC/Java uses an underlying automatic theorem prover to precisely reason about whether or not these kinds of errors can occur.

A static program checker may perform *modular checking*, which means the checker's analysis can be performed on one program module at a time, for some appropriate notion of "module". Such a checker relies on annotations that specify the module interfaces. For example, type checkers use type annotations that specify method signatures, and ESC/Java uses light-weight pre- and postconditions that specify a contract between the callers of a method and the method's implementation. While the annotation burden of common type checkers seems acceptable to programmers, our experience with advanced checkers like ESC/Java suggests the additional annotation burden is the major obstacle in the adoption of the checker.

To help overcome this obstacle, we have developed an annotation inference algorithm for ESC/Java called Houdini [FL01, FJL01]. Houdini works by first guessing a large set of *candidate annotations*, which are generated based on heuristics about what annotations might be useful

in reasoning about the program's behavior. Many of these guessed annotations will be invalid. Houdini then repeatedly calls ESC/Java and removes any candidate annotations that ESC/Java determines to be in conflict with the program. Eventually, the set of remaining annotations are valid for the program, and the algorithm terminates.

Houdini has proven quite useful in inferring suitable annotations for ESC/Java. It is capable of inferring many annotations that would otherwise have to be supplied by the programmer. Our experience suggests that these inferred annotations are useful in statically detecting software defects. In particular, ESC/Java produces many fewer warnings on the Houdini-annotated program than on the original unannotated program [FL01]. However, a significant limitation of the original Houdini algorithm is that it is quite slow.

In this paper, we describe an optimized Houdini algorithm that yields an order of magnitude performance improvement over the previous algorithm. The sources of this performance improvement include:

- We avoid calling ESC/Java from Houdini's inner loop, and instead call ESC/Java's underlying theorem prover directly.
- We distribute the refutation task over multiple processors, potentially on different machines.
- We carefully choose which method to check at each stage of the algorithm.

The rest of the paper is organized as follows. Section 2 defines a miniature extended static checker and outlines some of its properties. Section 3 reviews the Houdini annotation inference algorithm. Section 4 presents an optimized, sequential version of the algorithm, and Section 5 introduces a distributed version of it. We include correctness arguments for both of these new algorithms. Section 6 discusses work list ordering heuristics. Section 7 describes our implementation of the optimized algorithm and discusses performance measurements. Sections 8 and 9 discuss related work and conclude.

Before going on, we introduce some notational conventions. We write $\mathcal{P}(X)$ to denote the power set of X. The expression $\{\, x \mid r(x) :: t(x) \,\}$ denotes the set of terms of the form $t(x)$ for all x satisfying the range expression $r(x)$. For $\mathcal{Q}$ denoting $\forall$ or any associative operator that is symmetric on the elements of $\{\, x \mid r(x) :: t(x) \,\}$ (for example, $\cup$), the expression $\langle\, \mathcal{Q}\, x \mid r(x) :: t(x) \,\rangle$ denotes the application of $\mathcal{Q}$ to the elements of $\{\, x \mid r(x) :: t(x) \,\}$. If the range expression is *true*, we may omit the "$\mid$ *true*".

2 Extended static checking

This section describes a miniature extended static checker (ESC) for the language of guarded commands. ESC operates on guarded command procedures annotated with preconditions and postconditions. The goal of ESC is to verify that these annotations are consistent with the program's code. In the rest of this section, we introduce guarded commands, define how ESC generates verification conditions that check the correctness of annotations, and outline several pertinent properties of the checker.

2.1 Guarded command language

The syntax of the guarded command language is shown in Figure 1. We briefly review its semantics in this section; a more complete description is available elsewhere [Dij76, Nel89, BvW98]. The language includes the usual statements for assignment, sequencing, non-deterministic choice, local

m	::=	Id	(Procedure names)
v	::=	Id	(Variable names)
w	::=	$v*$	(Lists of program variables)
e	::=	$Expr$	(Boolean and arithmetic expressions)
a	::=	$Ann \mid Loc$	(Annotations or Location labels)
p	::=	**proc** m **is** S **end**	(Procedure declarations)
S	::=		(Statements)
		assert a: e | **assume** a: e | **var** w **in** S **end**	
	|	$v := e$ | $S_1 \,;\, S_2$ | $S_1 \,\square\, S_2$ | **call** m	

Figure 1: The guarded command language

variable declaration, and procedure invocation. An **assert** statement checks its condition and proceeds if the condition is satisfied or halts in the error state ("goes wrong") otherwise. An **assume** statement also checks its condition and proceeds if the condition is satisfied. If an assumed condition is not satisfied, then the computation "diverges silently", meaning the rest of the computation is not analyzed. For the purposes of static analysis, we distinguish two kinds of **assert** and **assume** statements: the ones labeled by program locations are written by the programmer and reside in the input program (for example, an **assert** statement may give the condition that a pointer is non-null and use a location label that indicates how to identify a failure of the condition to a user); the ones labeled by annotations are generated internally by our checker (see below). The guarded command language is sufficiently powerful to encode many high-level constructs, including those of the Java programming language [LSS99].

2.2 Annotations

To facilitate static analysis, procedures can be annotated with preconditions and postconditions. A precondition has the form $(\mathbf{pre}, m, e)$, and specifies that the boolean expression e hold in the pre-state of every invocation of procedure m. For a postcondition, which has the form $(\mathbf{post}, m, e)$, each variable v in the boolean expression e is written as v‘, to refer to the value of v in the pre-state of m, or as v', to refer to the value of v in the post-state of m. The postcondition specifies that the pre-post state relation e hold of every invocation of the procedure m. For example, the postcondition $(\mathbf{post}, m, v\text{‘} = v')$ stipulates that procedure m does not modify v. Callers are responsible for establishing preconditions, and procedure implementations postconditions.

Given a procedure and a set of annotations, ESC statically determines which of these annotations are *invalid* (may be violated by some execution) and returns the remaining set of valid annotations. Let us illustrate this process by an example:

proc f **is**	**proc** g **is**	$a_1 = (\mathbf{pre}, f, x < 0)$
$x := 0$;	**assert** l: $x > 0$;	$a_2 = (\mathbf{post}, f, x' > 0)$
call g	$x := 1$	$a_3 = (\mathbf{pre}, g, x \neq 0)$
end	**end**	$a_4 = (\mathbf{post}, g, x' > 0)$

This sample program consists of two procedures and four associated annotations. When checking the procedure f, ESC reports a_3 as invalid, because g's precondition is not established. When checking g, on the other hand, ESC has no complaints about the annotations, because those executions that reach the end of g without going wrong meet the postcondition a_4.

The checker can thus be represented as the function $ESC \in Proc \times \mathcal{P}(Ann) \rightarrow \mathcal{P}(Ann)$, where *Proc* is the set of procedure declarations, and *Ann* is the set of all annotations. For the example above,

$$\begin{array}{lcl} ESC(f, \{a_1,\ a_2,\ a_3,\ a_4\}) & = & \{a_1,\ a_2,\ a_4\} \\ ESC(g, \{a_1,\ a_2,\ a_3,\ a_4\}) & = & \{a_1,\ a_2,\ a_3,\ a_4\} \end{array}$$

In the remainder of this section we formalize this definition of *ESC*.

2.3 Translating procedures and call statements

ESC checks a procedure m by converting it to a logical formula called the *verification condition*. If this verification condition is valid, then any invocation of m that satisfies m's preconditions will not go wrong and will terminate only in states satisfying m's postconditions.

ESC generates verification conditions in two stages. In the first stage, ESC replaces each called procedure by that procedure's specification. This stage is defined by functions *Tp* and *Ts* that translate procedure declarations and individual statements respectively.

2.1 Definition: The functions $Tp \in Proc \times \mathcal{P}(Ann) \rightarrow Stmt$ and $Ts \in Stmt \times \mathcal{P}(Ann) \rightarrow Stmt$, where *Stmt* is the set of statements, are defined as follows:

$$\begin{array}{lcl} Tp(\textbf{proc}\ m\ \textbf{is}\ S\ \textbf{end}, A) & = & \langle\ ;\ a, e \mid a = (\textbf{pre}, m, e) \in A\ ::\ \textbf{assume}\ a\colon\ e\ \rangle\ ; \\ & & \textbf{var}\ V\text{`}, V' \ \textbf{in} \\ & & \quad \langle\ ;\ v \mid v \in V\ ::\ v\text{`} := v\ \rangle\ ; \\ & & \quad Ts(S, A)\ ; \\ & & \quad \langle\ ;\ v \mid v \in V\ ::\ v' := v\ \rangle\ ; \\ & & \quad \langle\ ;\ a, e \mid a = (\textbf{post}, m, e) \in A\ ::\ \textbf{assert}\ a\colon\ e\ \rangle \\ & & \textbf{end} \\ \\ Ts(\textbf{call}\ m, A) & = & \langle\ ;\ a, e \mid a = (\textbf{pre}, m, e) \in A\ ::\ \textbf{assert}\ a\colon\ e\ \rangle\ ; \\ & & \textbf{var}\ V\text{`}, V' \ \textbf{in} \\ & & \quad \langle\ ;\ v \mid v \in V\ ::\ v\text{`} := v\ \rangle\ ; \\ & & \quad \langle\ ;\ a, e \mid a = (\textbf{post}, m, e) \in A\ ::\ \textbf{assume}\ a\colon\ e\ \rangle\ ; \\ & & \quad \langle\ ;\ v \mid v \in V\ ::\ v := v'\ \rangle \\ & & \textbf{end} \\ \\ Ts(S, A) & = & map(\langle\, \lambda Q\ ::\ Ts(Q, A)\ \rangle, S) \qquad \text{if } S \text{ is not a } \textbf{call} \text{ statement} \end{array}$$ □

In this definition, V refers to the unprimed version of the variables mentioned in the postconditions of m, and V‘ and V' refer to the corresponding adorned variables. Here and in several other places in this paper, we use an informal map notation in which function *map* in the expression $map(f, S)$ applies f recursively to the sub-statements of S and reconstructs the top-level statement from the obtained results. Observe that the **assert** and **assume** statements created by *Tp* and *Ts* are labeled with their corresponding annotations. Applying *Tp* to the declarations of procedures f and

g from the example above results in the following two statements.

```
assume a1: x < 0 ;
var x`, x' in
  x` := x ;
  x := 0 ;
  assert a3: x ≠ 0 ;
  var x`, x' in
    x` := x ;
    assume a4: x' > 0 ;
    x := x'
  end
  x' := x ;
  assert a2: x' > 0
end
```

```
assume a3: x ≠ 0 ;
var x`, x' in
  x` := x ;
  assert l: x > 0 ;
  x := 1 ;
  x' := x ;
  assert a4: x' > 0
end
```

2.4 Generating verification conditions

Verification conditions are first-order logic formulas built up from the constants *false* and *true*, atomic boolean program expressions such as equality and inequality relations between program variables, the usual boolean connectives, and universal quantification. Additionally, we allow a formula e to be labeled by an annotation or program location a, yielding the labeled formula (**label** a: e). While these labels do not change the meaning of the underlying formula, they provide information to the subsequent stages of our checker. We will identify the syntactic class of formulas by *Formula*.

Verification conditions are computed using *weakest preconditions* [Dij76]. The weakest precondition of a statement S with respect to a postcondition R is the formula that characterizes those initial states from which the execution of S does not go wrong and terminates only in states satisfying R. The weakest precondition translation $wp \in \mathit{Stmt} \times \mathit{Formula} \to \mathit{Formula}$ is:

S	$wp(S, R)$	
$v := e$	$R(v := e)$	
assert a: e	$\begin{cases} (\textbf{label}\ a{:}\, e) \wedge R & \text{if } a \text{ is an annotation} \\ e \Rightarrow R & \text{if } a \text{ is a program location} \end{cases}$	
assume a: e	$e \Rightarrow R$	
$S_1 \,;\, S_2$	$wp(S_1, (wp(S_2, R))$	
$S_1 \,\square\, S_2$	$wp(S_1, R) \wedge wp(S_2, R)$	
var w **in** S **end**	$\langle\, \forall\, w \,::\, wp(S, R) \,\rangle$	provided the variables in w are not free in R

The verification condition of a procedure with respect to a set of annotations is defined by the function $VC \in \mathit{Proc} \times \mathcal{P}(\mathit{Ann}) \to \mathit{Formula}$ as follows:

$$VC(\textbf{proc}\ m\ \textbf{is}\ S\ \textbf{end}, A) = wp(Tp(\textbf{proc}\ m\ \textbf{is}\ S\ \textbf{end}, A), \mathit{true})$$

The verification conditions, call them VC_f and VC_g, for the sample procedures f and g shown above are, after applying the substitutions and suitably renaming bound variables to avoid capture:

$$VC_f = \quad x < 0 \Rightarrow \langle\, \forall\, x`, x' \,::\, (\textbf{label}\ a_3{:}\, 0 \neq 0) \wedge \langle\, \forall\, x“, x'' \,::\, x'' > 0 \Rightarrow (\textbf{label}\ a_2{:}\, x'' > 0) \,\rangle\rangle$$
$$VC_g = \quad x \neq 0 \Rightarrow \langle\, \forall\, x`, x' \,::\, x > 0 \Rightarrow (\textbf{label}\ a_4{:}\, 1 > 0) \,\rangle$$

2.5 Checking verification conditions

The verification condition VC_f is not valid; this indicates that an invocation of f may violate some annotation. But checking the validity of VC_f does not reveal *which* annotation may be violated. Identifying the invalid annotations requires a mechanism of exposing a labeled subformula in a verification condition. The function $expose \in Ann \times Formula \to Formula$ is defined as follows:

$$\begin{array}{lcll} expose(a, (\textbf{label } b{:}\, e)) & = & \begin{cases} e, & \text{if } b = a \\ true, & \text{otherwise} \end{cases} & \\ expose(a, R) & = & map(\langle\, \lambda Q \,::\, expose(a, Q)\, \rangle, R) & \text{if } R \text{ is not a labeled formula} \end{array}$$

We say that a formula R *refutes* an annotation a if $expose(a, VC_f)$ is not valid. Thus, after some rudimentary simplifications, we see that VC_f refutes a_3:

$$\begin{array}{lcll} expose(a_1, VC_f) & = & true & \\ expose(a_2, VC_f) & = & x < 0 \Rightarrow \langle\, \forall\, x\text{`}, x' \,::\, \langle\, \forall\, x\text{``}, x'' \,::\, x'' > 0 \Rightarrow x'' > 0\, \rangle\rangle & \\ expose(a_3, VC_f) & = & x < 0 \Rightarrow \langle\, \forall\, x\text{`}, x' \,::\, 0 \neq 0\, \rangle & \leftarrow \text{Invalid!} \\ expose(a_4, VC_f) & = & true & \end{array}$$

We now have all the necessary ingredients to complete the formalization of the extended static checker.

2.2 Definition: The extended static checker $ESC \in Proc \times \mathcal{P}(Ann) \to \mathcal{P}(Ann)$ is defined by the equation

$$ESC(p, A) = \{\, a \mid a \in A \wedge [expose(a,\, VC(p, A))] \,::\, a \,\}$$

where the square brackets represent the validity testing operator. □

The checker invocation $ESC(p, A)$ returns the set of annotations in A not refuted by p. We say that a set A of annotations is *valid* for a program (set of procedure declarations) P, written $P \vdash A$, if $ESC(p, A) = A$ for all $p \in P$.

The function ESC is contractive and monotonic [FJL01]; we will exploit these properties in the development of annotation inference algorithms. These algorithms also rely on the notion of a procedure being independent of a set of annotations. We say that an annotation $a \in Ann$ is *assumed* by a procedure p if a is a precondition of p or a postcondition of some procedure called by p, and a is *asserted* by p if a is a precondition of some procedure called by p or a postcondition of p. A procedure p is *independent* of a set of annotations $A \subseteq Ann$, written $indep(p, A)$, if A does not contain annotations assumed by p. The independence relation enjoys the following three properties, for any procedure $p \in Proc$ and annotation sets $A, B \subseteq Ann$ [FJL01]:

$$indep(p, A) \Rightarrow ESC(p, A \cup B) \subseteq A \cup ESC(p, B) \qquad \text{(Ind0)}$$
$$indep(p, \emptyset) \qquad \text{(Ind1)}$$
$$indep(p, A) \wedge indep(p, B) \Rightarrow indep(p, A \cup B) \qquad \text{(Ind2)}$$

ESC/Java is an implementation of the approach outlined in this section for the full Java programming language. Our initial hope for ESC/Java was that it would serve to improve the state of software engineering, in terms of both the development time and the reliability of the resulting system. However, although ESC/Java works well on annotated programs, it has not achieved our intended goals, mainly because catching defects in legacy, unannotated programs using ESC/Java is

$A := G$;
$W := P$;
while $W \neq \emptyset$ **do**
  pick p **such that** $p \in W$;
  $A' := ESC(p, A)$;
  if $A' = A$ **then**
    $W := W \setminus \{p\}$
  else
    $W := W \cup \{\ f \mid f \in P \wedge \neg indep(f, A \setminus A') :: f\ \}$;
    $A := A'$
  endif
endwhile

Figure 2: The basic Houdini algorithm, which computes A to be the largest subset of G that is valid for program P.

an arduous process. It is possible to run ESC/Java on an unannotated program, but this produces an excessively large number of false alarms. Alternatively, one can manually insert appropriate annotations into the program, but this is a very time-consuming task for large programs. Preliminary experience with ESC/Java indicates that a programmer can annotate an existing, unannotated program at the rate of a few hundred lines per hour, or perhaps at a lower rate if the programmer is unfamiliar with the code.

3 The Houdini annotation inference algorithm

To enable more cost-effective use of ESC/Java, we developed an annotation assistant for ESC/Java, called Houdini [FJL01]. The Houdini algorithm starts by generating a finite set G of *candidate annotations*. This set is generated from the program text based on heuristics about what annotations might be useful in reasoning about the program's behavior. For example, since a common precondition in manually-annotated programs is that an argument of reference type is non-null, the candidate annotation set includes all preconditions of this form. Other useful heuristics for guessing candidate annotations are described elsewhere [FL01].

Many of the annotations in the candidate set G will of course be incorrect. We would like to find the largest subset of G that is valid for the program P. In an earlier paper, we show that there exists a unique maximal subset of G that is valid for P [FJL01].

The algorithm shown in Figure 2 provides a means to compute this unique maximal subset of G. The algorithm starts with the annotation set G and removes annotations from it until a valid subset is reached. The algorithm maintains a work list W that contains the procedures that are still unchecked with respect to the current set of annotations A. When checking a procedure p from W, any annotations in A that are not valid for p are removed from A, and the work list is extended with the procedures that assume any of the refuted annotations. The algorithm terminates when the work list becomes empty, and at this point A is the largest valid subset of G.

The algorithm computes A as the largest valid subset of G, that is, it establishes:

$$A \subseteq G \tag{AA0}$$
$$P \vdash A \tag{AA1}$$
$$\langle\, \forall\, B \mid B \subseteq G :: P \vdash B \;\Rightarrow\; B \subseteq A \,\rangle \tag{AA2}$$

which has a short proof [FJL01].

The Houdini algorithm is actually what might be termed a two-level analysis. It uses ESC to perform precise procedure-local reasoning based on the underlying automatic theorem prover. Houdini's intraprocedural analysis is an abstract interpretation [CC77] based on the power set lattice generated by the candidate annotation set G. It is easy to tune Houdini's analysis by choosing the initial candidate set appropriately. However, performing a sufficiently precise analysis requires guessing a large number of candidate annotations, including preconditions and postconditions concerning alias relations (*e.g.*, $p \neq q$), non-nullness (*e.g.*, $p \neq null$), linear inequalities (*e.g.*, $0 \leq j$ or $j \leq a.length$), object invariants (*e.g.*, $\langle \forall p :: p\ instanceof\ Vector \Rightarrow \ldots \rangle$), and possibly other universally quantified annotations (*e.g.*, $\langle \forall j :: 0 \leq j \wedge j \leq a.length \Rightarrow a[i] \neq null \rangle$). Given a sufficiently complete candidate annotation set, the Houdini algorithm can perform a very accurate analysis. Unfortunately, the basic Houdini algorithm is quite slow, so it is difficult to apply it to large programs and large candidate annotation sets. The following sections describe improvements to the basic Houdini algorithm that significantly improve its performance and scalability.

4 Guarded verification conditions

When refuting annotations, the Houdini algorithm may invoke ESC to check each procedure multiple times. Each invocation of ESC involves generating a procedure's verification condition and checking its validity. The former operation seems redundant since the structure of the verification condition is derived from the code of the procedure, which remains constant. The current set of annotations A, on the other hand, is variable, but it affects the verification condition only in a limited way.

To illustrate this idea, we consider the verification conditions for the sample procedures f and g from Section 2. For the original set of four annotations $A = \{a_1, a_2, a_3, a_4\}$, the verification conditions are:

$$VC(f, A) = x < 0 \Rightarrow \langle \forall x`, x' :: \text{(}\mathbf{label}\ a_3 \text{:}\ 0 \neq 0\text{)} \wedge \langle \forall x``, x'' :: x'' > 0 \Rightarrow (\mathbf{label}\ a_2 \text{:}\ x'' > 0) \rangle \rangle$$
$$VC(g, A) = x \neq 0 \Rightarrow \langle \forall x`, x' :: x > 0 \Rightarrow (\mathbf{label}\ a_4 \text{:}\ 1 > 0) \rangle$$

The highlighted subformulas correspond to the invalid annotation a_3: it is asserted in f and assumed in g. Once the annotation inference algorithm refutes a_3, it must recompute the verification conditions with respect to the updated set of annotations $A' = \{a_1, a_2, a_4\}$:

$$VC(f, A') = x < 0 \Rightarrow \langle \forall x`, x' :: \langle \forall x``, x'' :: x'' > 0 \Rightarrow (\mathbf{label}\ a_2 \text{:}\ x'' > 0) \rangle \rangle$$
$$VC(g, A') = \langle \forall x`, x' :: x > 0 \Rightarrow (\mathbf{label}\ a_4 \text{:}\ 1 > 0) \rangle$$

These verification conditions are similar to the original ones except that the highlighted subformulas are deleted. Thus, the second pair of verification conditions could be obtained from the first pair by substituting *true* for the parts associated with the refuted annotation a_3 and leaving parts associated with the other annotations unchanged.

The example above motivates the following general approach. Let us generate a single "template condition" for every procedure in the beginning of annotation inference. When a procedure needs to be checked, we convert its template condition to an appropriate verification condition by replacing the parts related to the refuted annotations with *true* and leaving the parts related to the remaining annotations unchanged. The rest of this section describes an implementation of this approach.

For each annotation $a \in Ann$, let new variable g_a be the *guard variable* associated with a. Consider the formula $g_a \Rightarrow F$. Substituting *true* for g_a in this formula makes it equivalent to just

F, while substituting *false* makes the formula equivalent to *true*. Based on this idea, we define the following translation for generating guarded verification conditions.

4.1 Definition: The *guarded weakest precondition* translation $gwp \in Stmt \times Formula \to Formula$ is as follows. (The definition highlights the two differences between gwp and wp.)

S	$gwp(S, R)$	
$v := e$	$R(v := e)$	
assert a: e	$(g_a \Rightarrow (\textbf{label}\ a{:}\, e)) \wedge R$	if a is an annotation
	$e \Rightarrow R$	if a is a program location
assume a: e	$(g_a \Rightarrow e) \Rightarrow R$	
$S_1 \,;\, S_2$	$gwp(S_1, gwp(S_2, R))$	
$S_1 \,\square\, S_2$	$gwp(S_1, R) \wedge gwp(S_2, R)$	
var w **in** S **end**	$\langle\, \forall\, w \,::\, gwp(S, R)\, \rangle$	provided w are not free in R

□

To convert a guarded verification condition into an ordinary verification condition, we substitute *false* for the guard variables associated with refuted annotations and *true* for the remaining guard variables. This is formalized by the function $dropGuards \in Formula \times \mathcal{P}(Ann) \to Formula$, defined as follows:

$dropGuards(g_a, A)$	=	$true$	if $a \in A$
$dropGuards(g_a, A)$	=	$false$	if $a \notin A$
$dropGuards(R, A)$	=	$map(\langle\, \lambda\, Q \,::\, dropGuards(Q, A)\, \rangle, R)$	if R is not a guard variable

Figure 3 shows the optimized Houdini annotation inference algorithm that exploits guarded verification conditions. The algorithm pre-computes a guarded verification condition of every procedure in advance and applies *dropGuards* to convert a guarded verification condition into a verification condition whenever a procedure needs to be checked. This algorithm is more efficient than the original algorithm presented in Section 3 since in practice the *dropGuards* operation is much faster than the generation of a verification condition from the program source.

To show that the presented algorithm is equivalent to its predecessor, it is sufficient to establish that both algorithms compute the same value of the current set of annotations A' in every iteration of the **while** loop.

In the following discussion, we use these abbreviations:

$valid(F, a) = [expose(F, a)]$
$gvalid(F_g, A, a) = [expose(dropGuards(F_g, A), a)]$

First, let us show that guarded verification conditions computed from an initial annotation set G are equivalent to guarded verification conditions computed from the current annotation set A.

4.2 Lemma: For any procedure p, initial set of annotations G, set of annotations $A \subseteq G$, annotation $a \in A$, and guarded formulas R_1 and R_2, the following implication holds:

$gvalid(R_1, A, a) = gvalid(R_2, A, a) \quad \Rightarrow$
$gvalid(gwp(\,Tp(p, G), R_1), A, a) = gvalid(gwp(\,Tp(p, A), R_2), A, a)$ □

Proof: The proof proceeds by induction on the structure of p. The main intuition in the proof is that the annotations occurring in G but not in A are effectively ignored by the validity checking function. □

$A := G$;
$W := P$;
foreach $p \in P$ **do**
  $GVC_p := gwp(Tp(p, G), true)$
endforeach ;
while $W \neq \emptyset$ **do**
  pick p **such that** $p \in W$;
  $vc := dropGuards(GVC_p, A)$;
  $A' := \{\, a \mid a \in A \wedge [expose(a, vc)] :: a \,\}$;
  if $A' = A$ **then**
    $W := W \setminus \{p\}$
  else
    $W := W \cup \{\, f \mid f \in P \wedge \neg indep(f, A \setminus A') :: f \,\}$;
    $A := A'$
  endif
endwhile

Figure 3: The optimized Houdini algorithm.

The next lemma shows that conventional and guarded verification conditions computed from the same statement are equivalent.

4.3 Lemma: For any statement S, a set of annotations $A \subseteq G$, annotation $a \in A$, and guarded formulas R_1 and R_2, the following implication holds:

$valid(R_1, a) = gvalid(R_2, A, a)$ implies $valid(wp(S, R_1), a) = gvalid(gwp(S, R_2), A, a)$ □

Proof: By a straightforward induction on the structure of S. □

The final lemma establishes the equivalence of conventional and guarded verification conditions; its direct consequence is the equivalence of the annotation inference algorithms in Figures 2 and 3.

4.4 Lemma: For any procedure p, initial set of annotations G, set of annotations $A \subseteq G$, and annotation $a \in A$, we have the following equality:

$valid(VC(p, A), a) = gvalid(GVC(p, G), A, a)$ □

Proof: Follows from Lemma 4.3 and Lemma 4.2. □

5 Distributed annotation inference

The most computationally intensive parts of the optimized algorithm are its validity checks. However, if multiple processors are available, it is possible to distribute these checking tasks across the available processors, so that many procedures can be checked simultaneously. Figure 4 outlines a

$A := G$;
$W := P$;
foreach $p \in P$ **do**
  $GVC_p := Gwp.(Tp.p.G).true$
endforeach ;
while $W \neq \emptyset \vee \langle\, \exists\, k \;::\; M_k \neq \langle\rangle \,\rangle$ **do**
  ▯ $M_i = \langle\rangle \wedge W \neq \emptyset \;\longrightarrow$
    pick p **such that** $p \in W$;
    $M_i := \langle \mathbf{solving}, p, A \rangle$;
    $W := W \setminus \{p\}$
  ▯ $M_i = \langle \mathbf{solving}, p, B \rangle \;\longrightarrow$
    $vc := dropGuards(GVC_p, B)$;
    $A' := \{\, a \mid a \in B \wedge [expose(a, vc)] \;::\; a \,\}$;
    $M_i := \langle \mathbf{done}, p, B, A' \rangle$
  ▯ $M_i = \langle \mathbf{done}, p, B, A' \rangle \;\longrightarrow$
    $M_i := \langle\rangle$;
    if $A' \neq B$ **then**
      $A := A \cap A'$;
      $W := \{p\} \cup W \cup \{\, f \mid f \in P \wedge \neg indep(f, A \setminus A') \;::\; f \,\}$
    endif
endwhile

Figure 4: Distributed annotation inference algorithm

distributed Houdini algorithm. We use a guarded non-deterministic choice operator to model arbitrary interleaving of computation on different processors (*c.f.* [CM88]). The state of each processor i is modeled by the value in the variable M_i, as follows:

$\langle\rangle$	the processor is idle
$\langle \mathbf{solving}, p, A \rangle$	the processor has been given procedure p to check with respect to A
$\langle \mathbf{done}, p, B, A' \rangle$	the processor has finished checking p with respect to B producing result A'

The correctness of the distributed algorithm follows from using as the loop invariant the conditions (AA0), (AA2), and (1):

$$\begin{array}{l} \langle\, \forall\, u \mid u \in P \setminus W \;:: \\ \qquad ESC(u, A) = A \\ \quad \vee \;\; \langle\, \exists\, i \;::\; M_i = \langle \mathbf{solving}, u, B \rangle \wedge A \subseteq B \wedge indep(u, B \setminus A) \,\rangle \\ \quad \vee \;\; \langle\, \exists\, i \;::\; M_i = \langle \mathbf{done}, u, B, A' \rangle \wedge A \subseteq B \wedge indep(u, B \setminus A) \,\rangle \,\rangle \end{array} \tag{1}$$

Note that (AA1) follows from invariant (1) and the negation of the loop guard.

To prove that the algorithm terminates requires reasoning about the number of processors verifying a given procedure and the number of busy processors. If $M_i = \langle \mathbf{solving}, p, B \rangle$ or $M_i = \langle \mathbf{done}, p, B, \_ \rangle$, we say that processor i has *weight* $|B|$ and is *assigned* procedure p. The *weight* of a procedure p is the sum of weights of processors assigned p plus the size of the current annotation set $|A|$ if p is also in the work list W. Let *Weight* be the sum of weights of all procedures in the

program, and let *Load* be the number of processors whose state is $\langle \textbf{solving}, \_, \_ \rangle$. The termination measure ($|A|$, *Weight*, $|W|$, *Load*) lexicographically decreases after every iteration of the loop, and hence the distributed algorithm terminates on all finite inputs.

6 Work list ordering heuristics

The algorithms presented so far are correct no matter how the **pick** p statements choose a p satisfying $p \in W$. However, how this choice is made can have a significant impact on performance. This section outlines some heuristics for ordering the procedures in the work list.

6.1 Fastest-first and slowest-first heuristics

Consider a program containing a procedure p_0 with a precondition a and containing two procedures p_1 and p_2 that each calls p_0 without establishing a. In such a scenario, analyzing either p_1 or p_2 will uncover the invalid precondition a, and we may improve the overall performance by preferring the procedure with the "faster" verification condition. Clearly, we cannot avoid analyzing procedures with "slow" verification conditions completely: sooner or later every procedure in the work list must be checked. Using this heuristic, we hope to reduce the number of slow checks.

This *fastest first* ordering heuristic is implemented by timing each verification task and associating with each procedure the amount of time it takes to check its verification condition. When the algorithm selects the next procedure for verification, it chooses the one that took the least time the last time it was analyzed.

A different strategy is to order the jobs by *slowest first*. This heuristic may be useful in a multi-processor setting, since it may allow slow jobs to get a "head start".

6.2 No overlap heuristic

Having implemented the distributed algorithm, we ran it on a large test case and noticed that most of the time at least two processors were assigned the same procedure. While seemingly contradictory, this situation is actually possible and likely to occur. It arises when a processor i is analyzing a procedure p while another processor refutes some annotation that is assumed by p. The Houdini algorithm then reinserts p into the work list and can assign it to an idle processor j before processor i finishes its verification of p.

One (preemptive) approach to implement this *no overlap* heuristic is to abort the older verification task since it is subsumed by the new one. (By monotonicity of *ESC*, the annotations refuted by the older task would also be refuted by the newer task.) This strategy may be a win if many of the annotations that the older task will refute have already been refuted by other jobs.

Another (non-preemptive) approach is to not pick procedures that are currently being checked. This strategy may be a win for example if the verification of p spends a lot of time before it starts analyzing those annotations that are not in the eventual fixpoint.

7 Implementation and experiments

We have implemented all of the presented ideas, except preemptive no-overlap, in Houdini, which infers annotations for ESC/Java. Houdini consists of three components. Two of these are components of ESC/Java: a verification condition generator (which we modified to produce guarded

benchmark	lines of code	classes	routines	Annotations candidate	Annotations valid	Warnings ESC/Java	Warnings Houdini
Java2Html	558	5	32	398	86	70	11
WebSampler	1875	14	127	6252	5061	252	41
PachyClient	10928	57	653	33062	6076	1325	525
Cobalt	36152	173	1157	28363	9246	3978	649

Table 1: The benchmark programs used for the performance numbers in this paper.

verification conditions) and a theorem prover (called Simplify). The third component is a driver program that implements the annotation inference logic.

Given a Java program, Houdini first generates an initial set of annotations and uses ESC/Java to produce guarded verification conditions for every procedure (method or constructor) of the program. The obtained guarded verification conditions are stored as text files on disk. The driver program contains two modules: the coordinator and the server. The coordinator remotely starts a fixed number of server processes and performs scheduling of verification tasks among them. Given a procedure name, a server loads its verification condition from disk, applies *dropGuards* and various peephole-like optimizations to it, sends the obtained formula to a local copy of Simplify, and forwards the results of the verification back to the coordinator. The coordinator and servers communicate using sockets.

Communication overhead of the outlined design is insignificant, and the running time is dominated by the theorem proving component. The coordinator process idles about 90% of the time waiting for replies from the servers. Each server process, in turn, spends between 5% to 10% of its time preparing the verification condition for the theorem prover; Simplify takes the rest of the time.

Our main experiments have been conducted on four input programs:

- Java2Html [Jav], a 500-line program that turns Java programs into color-coded HTML pages,
- WebSampler, a 2,000-line program that performs statistical samplings of trace files generated by the web crawler Mercator [HN99],
- PachyClient, the 11,000-line graphical user interface of the web-based email program Pachyderm [Pac97], and
- "Cobalt", a proprietary 36,000-line program.

Table 1 shows some statistics about these programs. The table also shows how many candidate annotations were guessed for these programs and how many of the candidate annotations were valid. The difference between these two numbers is how many annotations Houdini refuted. For WebSampler, most of the candidate annotations remain, which is because the files we used included a lot of code that was not reachable from the given program entry points. Note, comparing PachyClient and Cobalt, that the number of candidate annotations is not just a function of program size but also of the particular declarations in the program.

The last two columns in Table 1 show how many warnings ESC/Java and Houdini produced on these programs. That is, they show how many warnings ESC/Java produced on the programs without any annotations and with the valid annotations inferred by Houdini.

benchmark	Generate cand. ann.	Original Houdini	New Houdini Gen. GVC	New Houdini P=1	New Houdini P=4
Java2Html	0:08	5:44	0:20	3:17	2:59
WebSampler	0:12	1:11:32	2:03	28:39	16:52
PachyClient	0:42	35:21:25	49:44	6:41:47	
Cobalt	4:41	60:00:00(*)	18:24	11:47:34	9:15:47

Table 2: Houdini analysis performance. All times are in hours:minutes:seconds. Due to some machine/network problems, numbers marked with (*) are approximations; we will rerun these experiments before a final version of this paper.

Ordering heuristic	P=1	P=4	P=4, no overlap
LIFO	13:00(*)		
FIFO	12:35		
Fastest first	11:48	9:16	
Slowest first	12:00(*)	17:49	7:00(*)
Random	13:38		

Table 3: Effect of work list ordering heuristics on the Cobalt benchmark. All times are in hours:minutes). Due to some machine/network problems, numbers marked with (*) are approximations; we will rerun these experiments before a final version of this paper.

Table 2 shows some performance numbers on the four benchmarks. The first number column shows that the time to generate the candidate annotations is insignificant. The next column shows the time the unoptimized Houdini required to refute the invalid candidate annotations.

The last three columns of Table 2 pertain to Houdini runs that use the optimizations described in this paper, in particular using the fastest-first heuristic (but not the no-overlap heuristic). The first of these shows the time required to generate the guarded verification conditions. Our current implementation performs this task on just one processor. It can be seen that this time is dominated by the time to refute candidate annotations, as shown in the other two columns for 1 and 4 processors, respectively.

Table 3 shows a comparison of different ordering heuristics for the Cobalt program. The no-overlap heuristic is a definite win, but we're less confident about recommending a clear winner among the other heuristics. We intend to run more experiments [even before a final version of this paper]. We guess that there may be other heuristics, perhaps hybrids of some heuristics we've described here, that would be even better.

8 Related work

Abstract interpretation [CC77] is a standard framework for developing and describing program analyses. We can view the Houdini algorithm as an abstract interpretation, where the abstract state space is the power set lattice $\mathcal{P}(G)$ and ESC is used to compute the abstract transition relation. As usual, the choice of the abstract state space controls the conservative approximations performed by the analysis. An advantage of the Houdini approach is that it is easy to tune these approximations by choosing the set of candidate annotations appropriately (provided that this set remains finite).

Note that ESC uses weakest preconditions for performing procedure-local analysis, which allows it to reason about sets of intermediate states that cannot be precisely characterized using Houdini's abstract state space $\mathcal{P}(G)$. Thus the two-level ESC+Houdini analysis may yield more precise results than a conventional single-level abstract interpretation that exclusively uses these abstract states to represent sets of concrete states.

The issue of annotation-based program checkers and associated annotation inference algorithms commonly arises in the study of type systems. In contrast to the Houdini framework, most type inference algorithms [Mit90, Car97, Mil78] do not reuse the type checker.

The Houdini algorithm can be viewed as a variant of predicate abstraction [GS97] in which each candidate annotation corresponds to a predicate. The Houdini algorithm finds the largest conjunction of these predicates that holds in all reachable states.

Daikon is a system that uses an empirical approach to find probable invariants [ECGN00]. These invariants are found by creating an instrumented version of the program that records a trace of intermediate program states, running the instrumented program on a test suite, and then analyzing the generated traces off-line to determine properties that hold throughout all runs. Given a suitably complete test suite, the inferred properties are likely to be true program invariants.

9 Conclusions

ESC/Java is a tool for detecting software defects early in the development cycle, when they are significantly cheaper to fix. ESC/Java has proven capable of catching a variety of software defects, but its reliance on the programmer to write annotations specifying procedure interfaces has hampered its application to large, unannotated programs.

The Houdini algorithm provides a means of inferring ESC/Java annotations automatically. Unfortunately, the original Houdini algorithm was quite slow and had difficulty scaling to large programs. This performance problem also limited our ability to improve the accuracy of Houdini by guessing additional candidate annotations.

In this paper, we have described optimizations that significantly improve Houdini's performance. These optimizations include:

- the use of guarded verification conditions, which eliminate the need to recreate verification conditions from scratch on each iteration of the algorithm;
- the distribution of the Houdini algorithm across multiple processors, allowing several procedures to be checked simultaneously; and
- a variety of heuristics for deciding which procedure to check next.

The combined effect of these optimizations is to improve the performance of Houdini by more than an order of magnitude.

This performance improvement has significant effects on the practicality of using Houdini and ESC/Java to catch software defects statically. The turnaround time of annotation inference is now much more tolerable. Within a given time limit (for example, a 16-hour overnight run), we can now analyze and detect software defects in significantly larger programs. Furthermore, we now have the performance headroom to further improve the accuracy of Houdini's analysis. In particular, we can extend Houdini to reason about additional kinds of properties by deriving heuristics for including the corresponding annotations in the candidate annotation set.

Acknowledgements

We'd like to thank Jim Saxe for helping us with many Simplify issues and Kent Dybvig for helping us with several Scheme issues.

References

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[Car97] Luca Cardelli. Type systems. *The Computer Science and Engineering Handbook*, pages 2208–2236, 1997.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), Limerick, Ireland*, June 2000.

[Ext] Extended Static Checking home page, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/esc/`.

[FJL01] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2001. To appear. (A preprint can be obtained from `research.compaq.com/SRC/personal/rustan/`.).

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of Formal Methods Europe (FME'01)*, March 2001. To appear. (A preprint can be obtained from `research.compaq.com/SRC/personal/rustan/`.).

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 72–83. Springer-Verlag, 1997.

[HN99] Allan Heydon and Marc A. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, December 1999.

[Jav] Java2html, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/software/`.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center, from `research.compaq.com/SRC/publications/`.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mit90] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 365–458. The MIT Press/Elsevier, 1990.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[Pac97] The Pachyderm email system, Compaq Systems Research Center. On the Web at `research.compaq.com/SRC/pachyderm/`, 1997.