Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lightweight automatic loop invariant selection

Bachelor Thesis

P. Pozdnyakov

Monday 30th September, 2019

Advisors: M. Eilers, Prof. Dr. P. Müller

Department of Computer Science, ETH Zürich

Abstract

One of the approaches used today in automatic program verification is deductive program verification. Alongside with specifications it uses auxiliary annotations to the program text to prove or refute desired properties of a program. The examples of such annotations are pre- and postconditions and loop invariants.

One of the possible ways to find the right annotations, which may help to verify a given program, is to guess them based on some heuristics. To find out which of the guessed annotations actually hold one could use the Houdini algorithm. One assumes first, that all annotations hold and runs the verifier. If some annotation is refuted by the verifier, it is deleted from the program text. The process repeats until no annotation is refuted.

In this work we concentrate our attention on the loop invariants. We show an alternative approach to find out which of the guessed invariants for the program to be verified hold without repeated calls to the verifier.

We present an encoding that allows us to infer invariants from the list of candidate invariants with consecutive verification of the given program using only one call to the verifier. We have also implemented the encoding in *C#* and tested it against Houdini on the number of programs. In most of the cases the running time of Houdini was as good or better as of our implementation. This is caused by the fact that in our implementation it is necessary to check exponentially many combinations of candidate invariants. We not exclude, however, that there still may exist a way to infer the invariants without exponential growth in running time complexity.

Contents

Chapter 1

Introduction

Gradually increasing complexity of modern software combined with the proliferation of applications used in life critical situations makes it desirable to be able to verify the correctness of a program and to be able to automate the process as much as possible.

A number of tools and approaches aiming to help to solve the problem of automatic program verification were developed in the past decades. Some of the tools available today are Viper [7], Boogie [6], Why3 [3] and ESC/Java [4].

The approach we are exploring in this work is *deductive program verification*, automated by using an SMT solver. It relies on specifying properties of a program through auxiliary annotations to the program text. These annotations may be based for example on the program specifications or may be guessed based on some heuristics. They help the verifier to prove whether certain properties of a program hold. The annotations include pre- and postconditions for methods as well as loop invariants.

Pre- and postconditions express the desired functionality of the method, but this still may not be sufficient to prove a certain program property. Finding the correct pre- and postconditions to a method may itself be rather challenging. The process of finding the right loop invariants seems to be even more involved. Difficulties in specifying annotations to a program text suggest that not only the verification process, but also the process of finding annotations should be automated.

We cannot, however, completely automate the process. Some annotations may rely on properties which are difficult to infer automatically.

In this case it is necessary to give a human user the possibility to influence the verification process by specifying such annotations by hand.

Houdini [5], one of the existing tools to help to find the right annotations for a program text, uses the so-called *guess-and-check* approach to infer pre- and postconditions for methods and loop invariants.

Based on manual inspection of annotated programs, heuristic rules are created to *guess* the right pre- and postconditions and loop invariants. These rules are then used to guess the candidate annotations to the given porgram. The candidate annotations are then automatically added to the program text. This extended program text is then given to a verifier, whose job is to *check* whether all of this annotations hold.

In the case of some annotation being invalid, as found by the verifier, it is removed from the program text. The verifier is then called again on this new program text.

The process continues until no annotation is refuted.

1.1 Project Overview

In this project we concentrate our attention exclusively on loop invariants. We want to explore alternative ways, compared to existing ones, of utilizing the guess-and-check approach for proving the program correctness.

We further restrict ourselves to investigate primarily the check-part of guess-and-check approach, i.e., we assume that a set of guessed loop invariants, *the candidate invariants*, is already given. For previously described reason we also want to support the possibility for a user to influence the verification process by specifying invariants, *the user invariants*, manually.

As opposed to guessed candidate invariants user invariants should hold, i.e., if some of the user invariants do not hold, the verifier should report it to the user. This report is useful, if the user is not only interested in successful program verification, but also wants to explicitly check, whether a certain invariant holds.

One of the potential problems with the check phase as it is currently employed, e.g., in Houdini [5], is that it usually has to be repeated several times until no annotation is refuted and the desired property of a program itself is proved or refuted.

Considering loop invariants, the extreme case would be to have a large amount of program code with only tiny fraction of it belonging to a loop statement, with the loop statement itself having a large number of invariants which do not hold. In the worst case Houdini would refute the invariants one by one, calling the verifier each time on the whole program text. The program text itself, however, except for the loop statement, remains unchanged in every call to the verifier. This means the verifier have to unnecessarily consider the same code chunks again and again.

In this thesis we give one possible way to transform the input program with the candidate invariants and the user invariants so that only one verifier call is necessary to infer invariants and to verify the program.

1.2 Project structure

In order to accomplish the goals of the project a number of examples of simple programs were considered. Based on these examples a number of different encoding approaches were developed. One of those was then chosen for the encoding.

The resulting encoding, as well as considerations about its limitations and problems during its development, is presented in Chapter 2. In Section 2.3 some thoughts on alternative approaches to the encoding are presented. These were not chosen for the current encoding either because they do not give sufficient gain in efficiency relative to implementation complexity, or they turned out to be unsound.

In order to test our ideas in practice, we have implemented the encoding as part of the Boogie verifier. A description of the implementation can be found in Chapter 3. Having an implementation at hand we made a comparison of running times of our implementation against Houdini. The description of results can be found in Chapter 4.

Final thoughts considering the results of the work as well as possible further research on the topic to be done in the future, are presented in *Conclusion*, Chapter 5.

Chapter 2

Encoding

The main part of this chapter is the definition of the transformation function we used in our encoding, Section 2.2. However, we first present the general ideas underlying the encoding, Section 2.1. At the end of the chapter, Section 2.3, we present some further thoughts and ideas related to our encoding and its potential alternatives.

2.1 General Idea

As stated in Section 1.2 we want to accomplish the following goals. Given a program to be verified, *the input program*, and the corresponding sets of candidate invariants and user invariants, we want to transform the program in a way that allows us to infer the loop invariants and to verify the program using only one call to the verifier. We also want the user to be notified, if some of the user invariants do not hold.

For the transformed program the same pre- and postconditions should hold as for the input program. For the loops of the transformed program the same invariants should hold as for the loops of the input program. At the same time the transformed program should allow us to infer invariants and verify the program in a way described above.

The main part of our encoding should be, therefore, the transformation function, explicitly defined in Section 2.2.

We define our encoding for the simple imperative language IMP [8]. The language structure is represented in the Table 2.1, where we denote vectors of variable names as $(x_1, ..., x_m)$ and $(y_1, ..., y_n)$, vectors of expres-

Table 2.1: IMP Structure

Procedure	*proc*	$procedure\ m(x_1, \ldots, x_m)\ returns(y_1, \ldots, y_n)\{s\}$
Statement	s	$skip \mid x := e \mid (x_1, ..., x_m) := call\ m(e_1, ..., e_k) \mid$ $s; s \mid if(b)\ then\ s\ else\ s \mid while(e)\ do\{s\} \mid$ $assume\ e \mid assert\ e \mid x := havoc \mid var\ x$
Expression	e	$e := c \mid x \mid e \oplus e$

sions as $(e_1, ..., e_k)$ and where $c \in \mathbb{R}$ and $\oplus = \{+, -, *, /, \%, ==>, <==, <==>, \vee, \wedge, \neg\}$.

The transformation function takes as arguments procedures and statements of the input program. It then transforms them appropriately, such that the resulting transformed program has the desired properties, described above.

Before we explain further, what transformations should be done to the input program, we first describe the input itself and the preprocessing we perform.

As input we have a program we want to verify and candidate and user invariants to be given for. In order to make our encoding more general, we allow different sets of candidate and user invariants be given for different loops. Hence, we assume that besides the program itself we have a mapping *Map* from a loop, represented through a unique *id*, and the tuples of two corresponding sets of candidate and user invariants as input.

As an implicit preprocessing step we transform the mapping as follows. We want all user invariants to hold. If it is not the case, we want it to be reported to the user. We therefore combine all user invariants defined for the same loop in one invariant, *the user invariant*, by conjoining them. As for the candidate invariants for a given loop, we have to consider each possible combination of them, i.e., the power set without an empty set, since, for example, two candidate invariants together may be an

invariant, but none of them separately is an invariant.

For each loop of the input program we now define the new set of candidate invariants as follows. We first take the power set P of the input candidate invariant set for that loop. We then take the union of each element of the power set and the user invariant. For each element of the resulting set P' we then conjoin all candidate invariants contained in this element to build one single candidate invariant. We call the resulting set of the candidate invariants P''.

We now define a new mapping Map' from the loop *id*s to the corresponding sets P'' of the candidate invariants. We do not need to handle user invariants separately anymore since the power set P also contains an empty set. By applying the union function to the empty set and the user invariant we get exactly one element representing the user invariant in P' and, hence, the same element in P''.

From now on, whenever we refer to a mapping from loop *id*s to the sets of candidate invariants we mean the mapping Map'. When we speak about a set of candidate invariants we mean the set P''. And when we speak about a candidate invariant, we speak about an element from the set P''.

In order to understand what transformations should be done to the input program, let us first consider, what should hold for a candidate invariant to be an invariant. The candidate invariant should hold immediately before the loop, at the beginning of the loop and at the end of the loop. Hence, to infer the invariants it must be possible to check these three properties for each candidate invariant. It should also be enough for our transformations to only affect the loops of the input program.

For each candidate invariant of a given loop we, therefore, need firstly to simulate the loop to see, whether three properties mentioned above are satisfied. We refer to the phase of simulating the loop to infer, whether a candidate invariant is an invariant, and its corresponding encoding as *the simulation loop*.

After the simulation loop is performed for each candidate invariant all the invariants are known. It is now possible to check whether the user invariants hold. In order to do so it is enough to assert the disjunction of all the elements of the candidate invariant set for the given loop. Since we incorporated the corresponding user invariant in each of the candidate invariants in a way described above, at least one of the candidate invariants has to hold. If it is not the case, i.e., the assertion is not satis-

fied, it indicates that at least one of the user invariants for that loop does not hold.

After all invariants for the given loop are inferred and the check whether all corresponding user invariants hold is performed, the actual verification phase of the loop starts. We refer to this phase and the corresponding encoding as *the original loop*. In order for the original loop to use the information about the previously inferred invariants, boolean flags are declared for each possible invariant. The simulation phase for a candidate invariant either sets the flag or does not.

The simulation loop and original loop are defined in a manner described above for each loop of the input program. The corresponding transformations surface to achieve the goals of the project.

Before we explicitly give the corresponding encoding in Section 2.2, we describe how the simulation loop phase and the original loop phase should be performed and what we need for this.

Having both the simulation and original loop means performing the same loop at least twice, whereas in the input program it is performed only once. We do not want to affect the values of variables used in the original loop while performing the simulation of that loop. Therefore, for every target variable of the input program we define a copy of it, *the duplicate*. We refer to variables that may change their value inside the corresponding loop body, e.g., through assignment to them, as target variables or targets. It suffices to define only one duplicate for each target variable. We use duplicates of the targets while simulating the loop and original target variables in the original loop.

Every simulation phase starts by checking whether the corresponding candidate invariant holds before the loop. For that purpose we define a boolean variable for each candidate invariant of each loop (as defined by our mapping Map'). This boolean flag is set to true, if and only if the candidate invariant holds before the loop.

To be able simulate an arbitrary iteration, we need to havoc the values of the duplicates for that loop. The only restrictions on the duplicates values are then given only by our assumptions about loop condition and the candidate invariant. If the previously defined boolean flag was set to true the loop condition is checked. If it may be satisfied the loop body is executed once, i.e., we simulate an arbitrary iteration of the loop, with restrictions put on the havocked duplicate values by the loop condition and the candidate invariant. After the simulation of of an arbitrary

iteration the check whether the candidate invariant hold is performed again. For that check the target variable names of the loop used in the candidate invariant should also be replaced with the corresponding duplicate names.

If the candidate invariant also holds after an arbitrary iteration it is actually an invariant. To store this information we define for each candidate invariant of each loop a boolean flag, which, when set to true, indicates that the corresponding candidate invariant is an invariant.

If the candidate invariant holds before the loop, but the loop condition may never be satisfied, it is trivially an invariant and the corresponding flag is set to true. In all other cases the flag indicating, whether the candidate invariant is an invariant is set to false.

The original loop phase is similar to the simulation loop phase. Here the original names of target variables are used instead of duplicates. After havocking the target variable values the assumption according to the previously set boolean flags that the inferred invariants hold is made. An arbitrary iteration of the loop is then performed. After assuming the negation of the loop condition the verifier continues with the program text below the loop.

We now continue with the explicit definition of the transformation function.

2.2 Transformation Function

2.2.1 Definitions

We first give a number of definitions listed in the Table 2.2, which will then be used through out the Section to improve the readability.

Table 2.2: Definitions

$Invs'$	The tuple built from the Map', in the following way. Recall that Map' as defined in Section 2.1 is a mapping between the loop id and the corresponding set of candidate invariants.

	Each element of each set is also an element of $Invs'$ and vice versa.
$LtoI(Invs', loop\_id)$	A function, which takes as arguments a tuple of candidate invariants and a loop id and outputs an ordered set of indices of candidate invariants, for the corresponding loop.
$loop$	$while(e)\ do\{s\}$
$get\_id(loop)$	A function, which returns the unique identifier of the $loop$.
$get\_nested\_id(outer\_loop\_id)$	A function, which returns the set of identifiers of the loops inside the $outer\_loop$.
$T(loop\_id)$	A function, which returns a tuple of names of all targets of a given loop.
$locals$	$\cup_{loop \in proc} T(loop\_id)$, where $proc$ is the method under transformation.
$vars$	$(x_1, x_2, ...)$, where $x_1, x_2, ...$ are variable names.
$fresh(vars)$	Variable names $x_1, x_2, ...$ are fresh, i.e., do not occur in the procedure under transformation.
$M(vars)$	A function, which returns a tuple of variables $vars'$ such that there is a bijection between $vars$ and $vars'$ and $fresh(vars')$.

$locals'$	$M(locals)$.
$\odot_{i=1}^{n} s_i$	The sequential composition of n statements $s_1, ..., s_n$.
$xs = (x_1, ..., x_m)$	The vector of parameter names.
$ys = (y_1, ..., y_n)$	The vector of return variable names.
$e[tuple_1 / tuple_2]$	Replacement of the variable names in $tuple_1$ with variable names in $tuple_2$ in expression e.

We further make the following three assumptions.

1. Input variables are immutable.
2. The statement $(x_1, \ldots, x_m) := call\ \mathrm{m}(e_1, ..., e_n)$ is equivalent to

   ```
   assert [preconditions]
   havoc (x1,...,xm)
   assume [postconditions]
   ```

   and is implicitly replaced by the latter in the input program before we start the transformation. We therefore do not define the transformation function for the statement $(x_1, \ldots, x_m) := call\ \mathrm{m}(e_1, ..., e_n)$.
3. The evaluation of an expression cannot change the program state.
4. The set of candidate invariants is not empty. In case there is no input candidate invariants and no input user invariants specified, we have a trivial user invariant and a trivial candidate invariant, the literal *true*.

While performing the transformations we differentiate between the simulation loops and the original loops, which were defined in Section 2.1. The former is used to infer which of the candidate invariants hold. The latter is used while performing verification of the input program with already inferred invariants.

We further differentiate between performing the transformations inside and outside of a loop, as well as *outer loops* and *inner loops*. Outer loops are outside of any other loop of the corresponding procedure. Inner loops are inside at least one other loop.

2.2.2 Transformations

We define two separate transformation functions,$[[\bullet]]_I$ and $[[\bullet]]_{I\_L}(targets)$. The former is used to perform the transformations outside of a loop or inside of an original loop. The latter is used to perform transformations inside of a simulation loop.

Procedure

By definition a procedure cannot be inside a loop, so only $[[\bullet]]_I$ is defined for it. Its only purpose is to declare the duplicates of all target variables of the procedure, as defined in Section 2.1. The same names for duplicates of the target variables are used through the whole procedure, so we declare them right at the beginning using $varx'$; statement.

```
[[procedure m(xs) returns(ys){s}]]_I =

  procedure m(xs) returns(ys)
  {
    ⊙_{x'∈locals'} var x';

    [[s]]_I
  }
```

We apply our transformation function $[[\bullet]]_I$ to each procedure of the input program. We then recursively call it on all of the statements of a given procedure.

Skip, Sequential Composition, If, Assume, Havoc, Assignment, Declaration

For all the statements except assertions and loops the two transformation functions behave in the same way. $[[\bullet]]_I$ is the identity transformation. $[[\bullet]]_{I\_L}(targets)$ replaces the names of target variables with the corresponding duplicate names, defined at the beginning of the transformed procedure.

Skip :

```
[[skip]]_I = skip

[[skip]]_{I_L}(targets) = skip
```

Since *skip* statement does not affect the program state, our transformation functions do nothing in this case.

SequentialComposition :

$[[s;s]]_I$ = $[[s]]_I$; $[[s]]_I$

$[[s;s]]_{I\_L}(targets)$ = $[[s]]_{I\_L}(targets)$; $[[s]]_{I\_L}(targets)$

If :

$[[\texttt{if(e) then s else s}]]_I$ = $\texttt{if(e) then } [[s]]_I \texttt{ else } [[s]]_I$

$[[\texttt{if(e) then s else s}]]_{I\_L}(targets)$ =

$\texttt{if } (e') \texttt{ then } [[s]]_{I\_L}(targets) \texttt{ else } [[s]]_{I\_L}(targets)$

where $e'_i = e_i[targets/M(targets)]$

Assume :

$[[\texttt{assume e}]]_I$ = assume e

$[[\texttt{assume e}]]_{I\_L}(targets)$ = assume $e[targets/M(targets)]$

Havoc :

$[[\texttt{x:=havoc}]]_I$ = x:=havoc

$[[\texttt{x:=havoc}]]_{I\_L}(targets)$ = x':=havoc

where $x' = M((x))$

Assignment :

$[[\texttt{x:=e}]]_I$ = x:=e

$[[\texttt{x:=e}]]_{I\_L}(targets)$ = x':=$e[targets/M(targets)]$

where $x' = M((x))$

Declaration :

$[[\texttt{Var x}]]_I$ = var x

$[[\texttt{Var x}]]_{I\_L}(\texttt{targets}) = \texttt{var x}'$

where $x' = M((x))$

Assert.

$[[\texttt{assert e}]]_I = \texttt{assert e}$

$[[\texttt{assume e}]]_{I\_L}(\texttt{targets}) = \texttt{skip}$

For the assertion statements the $[[\bullet]]_I$ function is again the identity transformation. The $[[\bullet]]_{I\_L}(targets)$ function, however, replaces an assertion with the *skip* statement.

Recall that we apply the $[[\bullet]]_{I\_L}(targets)$ function in the simulation loop body, where the invariants are unknown. At this point in the program, the verifier is checking the loop body for all possible candidate combinations, including ones that do not contain candidates that actually hold. Therefore it may not be able to prove assertions that actually hold, and could be proven to hold for the right choice of candidates.

This leads us to consideration that "assert" statements should be replaced by "skip" statements for the process of invariants selection and added again to the program text for the original loop phase, when the invariants are known.

While.

Since the transformations made to the loops are rather complex, we add comments in the style of the *C* programming language directly into the transformation formula where appropriate. Curly braces are added to the encoding to improve readability and denote blocks, as in the *C* programming language.

We first give a step by step description of what the transformation functions do and then provide a corresponding formula.

$[[while(e)\ do\{s\}]]_I$. As discussed in Section 2.1 we only apply the function $[[\bullet]]_I$ either to an outer loop, or to the loop body in the original loop.

Firstly, the boolean variables to infer whether a candidate invariant holds are declared. For each candidate invariant from the outer loop and for each candidate invariant of all of its inner loops two such boolean variables are declared. One is the flag which is set to *true* iff the candidate invariant holds before the loop. The other is the flag which is set to true if the candidate invariant is actually an invariant at the end of the corresponding simulation loop, where we have already inferred this information.

In the next step for each candidate invariant of the outer loop the check is performed, whether the candidate invariant holds before the loop. The corresponding flags are then set accordingly.

For each candidate invariant a simulation loop, i.e., a simulation of an arbitrary iteration of the outer loop, is defined in the following way. A conditional statement to check whether the flag indicating that the candidate invariant holds before the loop is set to true is added to the program text. The else clause means the flag is set to false, i.e., the candidate invariant is definitely not an invariant and we may proceed to the next candidate. For the then clause the following transformations are performed.

For each target variable name of the corresponding loop of the input program we replace this name in the loop body with the name of the corresponding duplicate variable, declared at the beginning of the procedure. The values of the target variables duplicates are then havocked.

Depending on the flag indicating whether the candidate invariant holds before the loop, assumptions on the variable values according to this invariant with target variables replaced by their duplicates are made.

The check of the loop condition is now performed. If the loop condition is satisfied, i.e., it is possible to enter the loop at least once, a simulation of an arbitrary iteration of the loop is performed. An assumption that the loop condition with replaced target variables holds is added to the porgram text and the transformation function $[[\bullet]]_{I\_L}(targets)$ is applied to the loop body with target variable names of the outer loop as the second argument.

It is now possible to check whether the candidate invariant is an invariant. If the candidate invariant with replaced targets has held before an arbitrary iteration and still holds after it, the corresponding flag is set to *true*.

After the flags are set that indicate, for each candidate invariant, if it is an invariant, we encode an arbitrary iteration of an actual loop, the original loop, with no replacement of target variables. This part of encoding is used for proving correctness of the loop body.

The values of all target variables of the loop are havocked and assumptions that previously inferred invariants hold are made. A new boolean variable is then declared and a conditional statement based on the value of this variable is added to the program text.

If the variable is true, the assumption that the loop condition holds is made. The transformation function $[[\bullet]]_I$ ia then applied on the loop body, as if it was outside of a loop, and the statement *assume false* is added at the end of the *then* branch. This last statement is an indication that this part of the program text still belongs to the loop body and the program text below it can only be reached if the loop condition is false. The assumption that the loop condition is false is made in the *else* branch.

Despite the *assume false* statement at the end of the *then* clause, we still needed it. Here the possible assertions, which were removed from the simulation loop body, are examined.

It is now possible to check, whether the user invariant, as defined in Section 2.1 holds. The corresponding assertion, as described in Section 2.1, is added to the program text before the original loop. It is the first place were we can perform this check, since only at this point we know all the invariants for the loop.

We now give the definition of the $[[\bullet]]_I$ function. We abbreviate $if\ (e)$ *then* $s1$ *else skip* as $if\ (e)\ s1$.

```
Let Id = get_id(while(e) do{s})
Let Is = LtoI(Invs', Id)
Let Ids = get_nested_id(get_id(Id))


// boolean variables to infer, whether an invariant holds
⊙_{i∈Is} var on_i;
⊙_{i∈Is} var on_b_i;

// boolean variables to infer,
// which invariants hold in nested loops of this outer loop;
⊙_{id∈Ids}⊙_{i∈LtoI(Invs',id)} var on_id_i;
⊙_{id∈Ids}⊙_{i∈LtoI(Invs',id)} var on_b_id_i;
```

```
// the simulation loop

// check whether an invariant holds before the loop
⊙_{i∈Is} assume inv_i <==> on_b_i;

// for every candidate invariant for that loop
// simulate an arbitrary iteration
⊙_{i∈Is}
{
    // if the candidate invariant does not hold
    // before the loop, it cannot be an invariant;
    // we then go to the next candidate
    if (on_b_i) then
    {
        ⊙_{x'=M((x)),x∈T(Id)} x':=havoc;
        assume on_b_i ==> inv_i[T(Id)/M(T(Id))];

        // we only simulate the outer loop,
        // if we can ever enter it
        if(e) then
        {
            assume e[T(Id)/M(T(Id))];

            // transformed loop body
            [[s]]_{I_L}(T(Id));
        }

        // infer, whether the i-th invariant holds
        assume inv_i[T(Id)/M(T(Id))] ==> on_i;
    }
}


// we now check wether the user invariant,
// as defined in Section 2.1, holds;
assert ∨_{i∈Is} on_i;


// the original loop

var star;
⊙_{x∈T(Id)} x:=havoc;
⊙_{i∈Is} assume on_i ==> inv_i;

if (star) then
{
    assume e;
```

```
        // if the loop body contains another loop,
        // we will do at this place the corresponding
        // simulation and original loop phases
        [[s]]$_{I}$;

        assume false;
    }
    else
    {
        assume ¬ e;
    }
```

where $\wedge_{i \in Is} fresh(on\_b_i)$, $\wedge_{i \in Is} fresh(on\_b_i)$, $\wedge_{id \in Ids} \wedge_{i \in LtoI(Invs',id)} fresh(on\_id_i)$, $\wedge_{id \in Ids} \wedge_{i \in LtoI(Invs',id)} fresh(on\_b\_id_i)$ and $fresh(star)$

$[[while(e)\ do\{s\}]]_{I\_L}$. As discussed in Section 2.1, the function $[[\bullet]]_{I\_L}$ is applied to the inner loop in the simulation loop phase.

The function $[[\bullet]]_{I\_L}(targets)$ performs almost the same transformations to the loop as $[[\bullet]]_I$. The differences are as follows.

The boolean flags to infer invariants were already declared by the function $[[\bullet]]_I$. This was done solely to improve readability by putting the declaration of all the flags on one place. The only new boolean variable is for the loop condition of the simulation loop. Firstly, its value is set, then all values of the target variable duplicates are havocked. Now it is still possible to use the value of the loop condition, even if the values of the variables used in it may be unknown.

We did not need this trick in the outer loop, since it was enough to evaluate the loop condition with original variables. In an inner loop we work exclusively with the duplicates, as long as we are in the simulation loop.

Another difference is that at the end of the simulation loop the function $[[\bullet]]_{I\_L}(targets)$ does only three things. It havocs the values of the target variables duplicates of the corresponding inner loop. It then puts an assumption, that negated loop condition holds, again with target variables replaced with their duplicates. At the end it assumes that previously inferred invariants hold.

There is no original loop part. For this part we may need to know the invariants for the outer loop as well as all the invariants of the inner

loops the loop under consideration is inside of. We do not have this knowledge at this point, however.

We now give the definition of the $[[\bullet]]_{I\_L}$ function.

```
Let id = get_id(while(e) do{s})
Let Is = LtoI(Invs', id)

// check, whether an invariant holds before the loop
⊙_{i∈Is} assume inv_i <==> on_b_id_i;

// check the inner loop condition,
// as described in Section 2.1
var lc_id;
assume e[targets/M(targets)] <==> lc_id;

// for each candidate invariant
// simulate an arbitrary iteration
⊙_{i∈Is}
{
    // if the candidate invariant does not hold
    // before the loop, it cannot be an invariant;
    // we then go to the next candidate
    if (on_b_id_i) then
    {
        ⊙_{x'=M((x)),x∈T(id)} x':=havoc;
        assume on_b_id_i ==> inv_i[T(id)/M(T(id))];

        // we only simulate the outer loop,
        // if we can ever enter it
        if(lc_id) then
        {
            assume e[T(id)/M(T(id))];

            // transformed loop body
            [[s]]_{I_L}(T(id));
        }

        // infer, whether the i-th invariant holds
        assume inv_i[T(id)/M(T(id))] ==> on_id_i;
    }
}


// the statements below put the neccessary restictions
// on the corresponding variable values,
// the restictions may be needed to infer invariants
// for the outer loop of this inner loop
```

```
⊙_{x'=M((x)),x∈T(id)} x' := havoc;
assume ¬e[targets/M(targets)];
⊙_{i∈Is} assume on_id_i ==> inv_i;
```

where $fresh(lc\_id)$ holds and on_id_i and $on\_b\_id_i$ for $i \in Is$ should be declared earlier

2.3 Potential Problems of the Encoding

2.3.1 Possible Inefficiencies

As one can see from the encoding in the previous section, the most changes to the initial program text are done when it comes to the while statements. These changes are also a source of potential inefficiencies.

One of the main differences between the two functions, $[[\bullet]]_I$ and $[[\bullet]]_{I\_L}(targets)$, lies in their handling of the while statement at the point, where the loop invariants for the corresponding loop are inferred.

In case of the function $[[\bullet]]_{I\_L}(targets)$ we negate the loop condition and assume that the previously inferred invariants hold. After that the simulation of the corresponding outer loop continues.

In case of the function $[[\bullet]]_I$ we proceed to the simulation of the actual loop, with original variable names.

Since the invariants for the outer loop at this point are known, we recursively apply the function $[[\bullet]]_I$ to the loop body, as if the loop body was a program text outside of any loop.

The recursion and, hence, the repeated inference of the invariants for the inner loops is needed, however. Although we have inferred invariants for the outer loop, we are not able to store the values of the flags for the inner loops invariants, which have helped us to infer the outer loop invariants.

This implies potential problems if the input program has nested loops. Although we have only one original loop phase for each loop of the input program, the simulation loop phase is repeated for any inner loop at least as many times, as there are loops in which the given loop is contained.

In the current encoding both functions, $[[\bullet]]_I$ and $[[\bullet]]_{I\_L}(targets)$, infer whether a candidate invariant holds separately for each candidate invariant, i.e., for each element of the power set of the input candidate invariants set. Therefore the simulation phase of the inner loop will be potentially repeated many more times than just the number of its respective outer loops, since it is the part of any simulation of these outer loops.

The size of the code parts responsible for the loop simulation grows, therefore, exponentially with the number of elements of all the input candidate invariant sets of this loop and its respective outer loops. This, of course, has negative consequences for the verification time.

2.3.2 Potential Solutions

The presented encoding of the loops is obviously inefficient because loop bodies are duplicated very often. An alternative idea would be to try out all possible combinations of candidates at once in a single loop simulation.

One way to do that would be to have an additional flag *on_a* for each candidate invariant in the outer loop and add an *assume on_a ==> inv* statement for each candidate invariant at the beginning of the simulation of an arbitrary iteration of the outer loop.

The corresponding, unsound, encoding for the $[[\bullet]]_I$ is presented below.

```
Let Id = get_id(while(e) do{s})
Let Ids = get_nested_id(Id)


[[while(e) do{s}]]_I =

    // boolean variables to infer,
    // whether an invariant holds
    ⊙_{i=1}^{k} var on_i;
    ⊙_{i=1}^{k} var on_b_i;
    ⊙_{i=1}^{k} var on_a_i;

    // boolean variables to infer,
    // which invariants hold in nested loops

    ⊙_{id∈Ids} ⊙_{i=1}^{k} var on_id_i;
    ⊙_{id∈Ids} ⊙_{i=1}^{k} var on_b_id_i;
    ⊙_{id∈Ids} ⊙_{i=1}^{k} var on_a_id_i;
```

```
// check, whether an invariant holds before the loop
⊙_{i=1,inv_j∈I'}^{k} assume inv_i <==> on_b_i;

// simulate an arbitrary iteration
⊙_{x'=M(x),x∈T(Id)} x':=havoc;
⊙_{i=1,inv_j∈I'}^{k} assume on_a_i ==> inv_i[T(Id)/M(T(Id))];

// we only simulate the outer loop, if we can ever enter
    it
if (e) then
{
    assume e[T(Id)/M(T(Id))];

    // transformed loop body
    [[s]]_{I_L}(T(Id));
}

// infer, which invariants hold
⊙_{i=1,inv_j∈Invs'}^{k} assume (on_b_i ∧ (on_a_i ==> inv_i[T(Id)/M(T(Id))
    ])) ==> on_i;

// actual loop
var star;
⊙_{x∈T(Id)} x:=havoc;
⊙_{i=1,inv_j∈Invs'}^{k} assume on_i ==> inv_i;

if (star) then
{
    assume e;

    [[s]]_{I_L};

    assume fale;
else
{
    assume ¬e;
}
```

where $fresh(on_1, ..., on_k) \wedge fresh(on\_b_1, ..., on\_b_k) \wedge fresh(on\_a_1, ..., on\_a_k) \wedge$
$\wedge_{id \in get\_nested\_id(Id)} fresh(on\_id_1, ..., on\_id_k) \wedge$
$\wedge_{id \in get\_nested\_id(Id)} fresh(on\_b\_id_1, ..., on\_b\_id_k) \wedge$
$\wedge_{id \in get\_nested\_id(Id)} fresh(on\_a\_id_1, ..., on\_a\_id_k) \wedge$
$fresh(star)$

The problem with this approach is that the combination of *assume on_a ==> inv* statements may lead to unexpected restrictions on the values of variables used in invariant expressions, rendering the inference process unsound.

We illustrate that with the following example.

Let us further consider the following Boogie program.

```
function lol(b: bool): bool;

procedure bar() returns ()
{
    var a: int;
    var b: int;
    var c: int;
    var temp: int;
    var stop : bool;
    stop := false;
    a := 1;
    b := 2;
    c := 3;
    while (!stop)
        invariant a != c;
        invariant a != b;
    {
        stop := lol(stop);
        temp := a;
        a := b;
        b := c;
        c := temp;
    }

    // should fail without b != c invariant,
    // should succeed otherwise.
    assert {:msg "outer"} a != c;
}
```

Let us now look at the result produced by our unsound encoding.

```
function lol(b: bool): bool;

procedure bar_transformed() returns()
{
    var a: int;
    var b: int;
    var c: int;
    var temp: int;
    var stop: bool;
    var stop_duplicate: bool;
```

```
var temp_duplicate: int;
var a_duplicate: int;
var b_duplicate: int;
var c_duplicate: int;
var anon0_on_1: bool;
var anon0_on_b_1: bool;
var anon0_on_a_1: bool;
var anon0_on_2: bool;
var anon0_on_b_2: bool;
var anon0_on_a_2: bool;
var anon0_on_3: bool;
var anon0_on_b_3: bool;
var anon0_on_a_3: bool;

var anon0_star: bool;

stop := false;
a := 1;
b := 2;
c := 3;
assume (a != b && a != c) <==> anon0_on_b_1;
assume a != c <==> anon0_on_b_2;
assume a != b <==> anon0_on_b_3;

havoc stop_duplicate;
havoc temp_duplicate;
havoc a_duplicate;
havoc b_duplicate;
havoc c_duplicate;
assume anon0_on_a_1 ==> (a_duplicate != b_duplicate &&
   a_duplicate != c_duplicate);
assume anon0_on_a_2 ==> a_duplicate != c_duplicate;
assume anon0_on_a_3 ==> a_duplicate != b_duplicate;

if (!stop)
{
    assume !stop_duplicate;
    stop_duplicate := lol(stop_duplicate);
    temp_duplicate := a_duplicate;
    a_duplicate := b_duplicate;
    b_duplicate := c_duplicate;
    c_duplicate := temp_duplicate;
}

assume (anon0_on_b_1 && (anon0_on_a_1 ==> (a_duplicate
   != b_duplicate && a_duplicate != c_duplicate))) ==>
   anon0_on_1;
assume (anon0_on_b_2 && (anon0_on_a_2 ==> a_duplicate !=
    c_duplicate) ==> anon0_on_2;
```

```
    assume (anon0_on_b_3 && (anon0_on_a_3 ==> a_duplicate !=
        b_duplicate) ==> anon0_on_3;

    havoc anon0_star;
    havoc stop;
    havoc temp;
    havoc a;
    havoc b;
    havoc c;
    assume anon0_on_1 ==> (a != b && a != c);
    assume anon0_on_2 ==> a != c;
    assume anon0_on_3 ==> a != b;

    if (anon0_star)
    {
        assume !stop;

        stop := lol(stop);
        temp := a;
        a := b;
        b := c;
        c := temp;

        assume false;
    }
    else
    {
        assume !!stop;
    }

    assert a != c;
}
```

Transformed Boogie Program

Assertion $a \neq c$ at the end of the transformed program is successfully proven. This should not be possible, however, without the invariant $b \neq c$.

The problem here is the following. Whenever the flag $anon0\_on\_1$ is set to *true*, i.e., we assume that the candidate invariant $a \neq c \wedge a \neq b$ at the beginning of the loop holds, it implies that the candidate invariants $a \neq c$ and $a \neq b$ also hold at the beginning of the loop and that the candidate invariant $a \neq c$ holds at the end of the loop. Therefore we are able to infer that the candidate invariant $a \neq c$ is an invariant. This should not be the case.

Although neither of candidate invariants holds by itself, the inference process for one candidate invariant affects the inference process for another one. This happens since both inference processes use the same variables.

This could be solved for example by introducing a new copy of target variables for each candidate invariant in the simulation loop, which would be equivalent to the sound encoding presented in this thesis. The number of variable copies grows exponentially in the number of input candidate invariants.

The way to solve the issue, which is used in the sound version of encoding and implementation, is to encode the simulation of the loop for each invariant separately. In this case the number of target variable copies remains constant, but the number of loop simulations grows exponentially in the number of input candidate invariants.

It seems, however, that there is now straightforward efficient solution to that problem.

Chapter 3

Implementation

In this chapter we give a brief overview of the implementation. We have chosen to implement our encoding, presented in the previous chapter, as an extension of the Boogie Programming Language. The implementation is done in *C#*, as is the implementation of Boogie itself.

Since the implementation follows rather strictly the encoding, we concentrate here mainly on the differences. The differences are caused mainly by the implementation of the Boogie Programming language itself, in particular the Boogie AST.

One particular difference is that we had to make the transformations not on the granularity of statements, as described in the encoding, but on the granularity of *BigBlock*s. *The BigBlock* is a unit element created by the parser. It contains of *the Simple Commands* and *the Structured Commands*, i.e., not control flow and control flow statements correspondingly. Each BigBlock always starts with a simple command, if there are any in program text, and always ends with one structured command, if there is any in program text. This implementation detail did not influence the semantics of the encoding, however.

The next two changes we have made to the encoding also have slightly changed the semantics. The changes, however, do not affect the process of invariants inference and the verification of the input program.

1. Boogie supports multiple left hand sides in an assignment statement. We have also implemented this functionality, which is not present in the encoding, in our extension.
2. The types in our encoding are implicit. If we declare a new variable, it has no indication of type. And if we declare a new variable

which is a copy of a target variable, it has the same type as the corresponding target variable; the type is, however, not explicitly given.

Our implementation supports all the types of the Boogie Programming Language.

In order for our implementation to read the input candidate invariants and the input user invariants they should be encoded in the input program text in the following way.

1. A candidate invariant *inv* is given by the statement *invariant* $\_b ==>$ *inv* for the corresponding loop. $\_b$ is a global constant, declared at the beginning of the input program text as *const* {: *existential true*} $\_b$: *bool*;.
2. A user invariant inv_u is given by the statement *invariant* inv_u for the corresponding loop.

This is the same format Houdini uses.

As the transformation phase reaches a loop, the corresponding candidate invariants set is built from previously read candidate invariants for that loop and user invariants for that loop as described in Section 2.1.

Chapter 4

Evaluation

We have evaluated the performance of our implementation against Boogie's Houdini implementation on various short programs, listed in the Appendix.

The processor we used was Intel(R) Core(TM) i7-6700HQ CPU, 2.60GHz, 4 Cores, 8 Logical Processors, with 16 GB RAM and 20GB Virtual Memory. The operating system was Windows 10 Pro, 64-bit version.

We ran our implementation and Houdini 10 times with each program as input. We then averaged the corresponding running times.

The programs used for evaluation may be divided into three categories.

1. The programs we used to develop and test the ideas for our encoding: *nested_loops_transformed_09.bpl* and *unsound_00.bpl*.
2. The programs to test the behavior of Houdini and our Implementation under certain conditions: *many_false_invariants.bpl* and *many_lines.bpl*. In both programs the amount of program test outside of any loop is much bigger then the amount of text inside a loop. In the first program there are nine candidate invariants all of which are not invariants. In the second program there are only two candidate invariants and they are also not invariants.

   The programs were written in such a way that Houdini can refute only one candidate invariant per one verifier call. They were supposed to help to understand, how much the verification time using Houdini changes relative to our Implementation with increasing number of candidate invariants, i.e., increasing number of verifier calls.

3. The real world applications: *kernel_structured_nocalls.bpl* and *HistogramAtomics.bp*
   These were taken from Accelerated parallel processing SDK [1] and translated to the Boogie Programming Language with GPUVerify [2].

   The translated programs were further rewritten by hand to replace *goto* statements with *while* statements and global variables with local ones. This last transformation was needed since our implementation does not count loops encoded with *goto*s as loops. We also needed to get rid of global variables because if some of them are modified inside a loop, our implementation would make this modification at least twice, at least once for the simulation loop phase and once for the original loop phase. This could be avoided by storing the values of global variables in auxiliary variables before performing a simulation of an arbitrary iteration of a loop and thereafter restoring these values. We did not provide this feature in our implementation, however.

The evaluation results are provided in Tables 4.1, 4.2, 4.3.

The first group of programs are rather short programs used to get a general idea of the performance of our implementation and Houdini. In all cases the running time of Houdini was smaller than that of our implementation. In this case the amount of code for loop simulations was bigger, than the amount of code outside of any loop. This lead us to the idea to evaluate the approach on programs where the amount of code for loop simulations would be much smaller then the code for the rest of the program.

Both of the programs of the second group have large amount of code outside of any loop and were written in such a way that Houdini can refute only one candidate invariant at a time. We see, however, that also in this case the verification times using Houdini are much shorter than verification times using our implementation. Performing exponentially many (in the number of the candidate invariants) simulations of a loop for both programs has a huge negative impact on running time also in the case where the code for the invariant inference phase is only a small fraction of the overall program text.

The third group of programs allowed us to test the behavior of our implementation in comparison with that of Houdini on real application programs. Also in this case running times of Houdini were smaller than those of our implementation.

Table 4.1: Evaluation Results

File Name	*nested_loops_transformed_09.bpl*	*unsound_00.bpl*
Number of Lines	34	35
Number of Lines Produced by Our Implementation	409	513
Running Time, Seconds Houdini	0.02	0.04
Running Time, Seconds Our Implementation	0.19	0.19
Number of Loops Total / Nested	2 / 1	1 / 0
Number of Invariants Correct / Incorrect	5 / 0	3 / 1

Table 4.2: Evaluation Results

File Name	*many_false_invariants.bpl*	*many_lines.bpl*
Number of Lines	450045	450031
Number of Lines Produced by Our Implementation	612331	600125
Running Time, Seconds Houdini	27.83	8.05
Running Time, Seconds Our Implementation	257.49	227.33
Number of Loops Total / Nested	1 / 0	1 / 0
Number of Invariants Correct / Incorrect	0 / 9	0 / 2

Table 4.3: Evaluation Results

File Name	*kernel_structured_nocalls.bpl*	*HistogramAtomics.bpl*
Number of Lines	316	236
Number of Lines Produced by Our Implementation	280810	9731
Running Time, Seconds Houdini	0.88	0.16
Running Time, Seconds Our Implementation	56.04	1.15
Number of Loops Total / Nested	2 / 1	1 / 0
Number of Invariants Correct / Incorrect	5 / 6	2 / 6

The long verification times using our implementation were expectable for programs where the amount of code inside a loop constitutes a significant amount of overall program text. Even if we have only one candidate invariant, our implementation already repeats the corresponding loop body twice. In the worst case, when this loop is the only program text we have, it already doubles the size of the program to be verified. The situation gets even worse if we have several candidate invariants.

Testing the running times on real world programs confirmed the previous thoughts. Since a loop body constitutes the significant part of the text of both programs longer running times of our implementation with respect to Houdini were expectable.

Evaluating the running times with the programs of the second group showed, however, that our implementation performance is affected by the duplicated loop bodies much more severely then we initially thought. Even with a tiny loop body and a huge amount of code outside of a loop our the running time of our implementation was much slower than that of Houdini. Although increasing the number of candidate invariants slowed down the verification time using Houdini, the difference in verification times using our implementation and Houdini remained big. Since we specifically used a very big amount of code outside of a loop, which should be bigger than that of an average real world program, it suggests that verification using Houdini is more efficient.

Chapter 5

Conclusion

In this thesis we presented an alternative approach of using the guess-and-check method to infer loop invariants and subsequently verify the input program.

We presented an encoding which allows to infer the loop invariants and verify a program using only one call to the verifier. This is fundamentally different from the corresponding approaches used today, e.g., Houdini. In the process of reducing the number of calls to the verifier we introduced, however, another problem. In our encoding we have to consider all possible combinations of the input candidate invariants for a given loop. This may significantly enlarge the amount of text of the transformed program leading to long verification times.

We implemented our encoding as an extension of the Boogie verifier and evaluated it against Houdini implementation for the Boogie Programming Language.

As we suggested, Houdini performed better than our implementation if the amount of code inside the loop body was significant relative to the overall amount of the program text. We found, however, that Houdini also performed better if the amount of code inside the loop body was negligibly small relative to the overall amount of the program text. This means that the problem of considering all possible combinations of the input candidate invariants for a given loop should be solved before our method can be used in practice.

As we discussed in Chapter 2, finding a solution to this problem means being able to infer several invariants simultaneously in such a way that inference of one invariant does not affect the inference of another.

The encoding as presented in this thesis may, however, still be useful not only as a proof of concept. Since for a given loop we perform the inference process for each of the possible combinations of the corresponding input candidate invariants separately and independently from one another, it suggests that our encoding can easily be parallelized.

Appendix A

Code Listings

A.1

```
type _SIZE_T_TYPE = bv32;

procedure _ATOMIC_OP32(x: [bv32]bv32, y: bv32) returns (z$1:
    bv32, A$1: [bv32]bv32, z$2: bv32, A$2: [bv32]bv32);


axiom {:array_info "$$in"} {:global} {:elem_width 32} {:
    source_name "in"} {:source_elem_width 32} {:
    source_dimensions "*"} true;

var {:race_checking} {:global} {:elem_width 32} {:
    source_elem_width 32} {:source_dimensions "*"}
    _READ_HAS_OCCURRED_$$in: bool;

var {:race_checking} {:global} {:elem_width 32} {:
    source_elem_width 32} {:source_dimensions "*"}
    _WRITE_HAS_OCCURRED_$$in: bool;

var {:race_checking} {:global} {:elem_width 32} {:
    source_elem_width 32} {:source_dimensions "*"}
    _ATOMIC_HAS_OCCURRED_$$in: bool;

var {:race_checking} {:global} {:elem_width 32} {:
    source_elem_width 128} {:source_dimensions "*"}
    _READ_HAS_OCCURRED_$$out: bool;


var {:race_checking} {:global} {:elem_width 32} {:
    source_elem_width 128} {:source_dimensions "*"}
    _ATOMIC_HAS_OCCURRED_$$out: bool;
```

```
const _WATCHED_OFFSET: bv32;

const {:global_offset_x} global_offset_x: bv32;

const {:global_offset_y} global_offset_y: bv32;

const {:global_offset_z} global_offset_z: bv32;

const {:group_id_x} group_id_x$1: bv32;

const {:group_id_x} group_id_x$2: bv32;

const {:group_size_x} group_size_x: bv32;

const {:group_size_y} group_size_y: bv32;

const {:group_size_z} group_size_z: bv32;

const {:local_id_x} local_id_x$1: bv32;

const {:local_id_x} local_id_x$2: bv32;

const {:num_groups_x} num_groups_x: bv32;

const {:num_groups_y} num_groups_y: bv32;

const {:num_groups_z} num_groups_z: bv32;

function {:bvbuiltin "bvadd"} BV32_ADD(bv32, bv32) : bv32;

function {:bvbuiltin "bvmul"} BV32_MUL(bv32, bv32) : bv32;

function {:bvbuiltin "bvult"} BV32_ULT(bv32, bv32) : bool;

procedure {:source_name "write_kernel"} {:kernel}
   $write_kernel($ni: bv32, $val: bv32, $nk: bv32);
  requires BV32_SGT(group_size_x, 0bv32);
  requires BV32_SGT(num_groups_x, 0bv32);
  requires BV32_SGE(group_id_x$1, 0bv32);
  requires BV32_SGE(group_id_x$2, 0bv32);
  requires BV32_SLT(group_id_x$1, num_groups_x);
  requires BV32_SLT(group_id_x$2, num_groups_x);
  requires BV32_SGE(local_id_x$1, 0bv32);
  requires BV32_SGE(local_id_x$2, 0bv32);
  requires BV32_SLT(local_id_x$1, group_size_x);
  requires BV32_SLT(local_id_x$2, group_size_x);
  requires BV32_SGT(group_size_y, 0bv32);
  requires BV32_SGT(num_groups_y, 0bv32);
```

```
  requires BV32_SGE(group_id_y$1, 0bv32);
  requires BV32_SGE(group_id_y$2, 0bv32);
  requires BV32_SLT(group_id_y$1, num_groups_y);
  requires BV32_SLT(group_id_y$2, num_groups_y);
  requires BV32_SGE(local_id_y$1, 0bv32);
  requires BV32_SGE(local_id_y$2, 0bv32);
  requires BV32_SLT(local_id_y$1, group_size_y);
  requires BV32_SLT(local_id_y$2, group_size_y);
  requires BV32_SGT(group_size_z, 0bv32);
  requires BV32_SGT(num_groups_z, 0bv32);
  requires BV32_SGE(group_id_z$1, 0bv32);
  requires BV32_SGE(group_id_z$2, 0bv32);
  requires BV32_SLT(group_id_z$1, num_groups_z);
  requires BV32_SLT(group_id_z$2, num_groups_z);
  requires BV32_SGE(local_id_z$1, 0bv32);
  requires BV32_SGE(local_id_z$2, 0bv32);
  requires BV32_SLT(local_id_z$1, group_size_z);
  requires BV32_SLT(local_id_z$2, group_size_z);
  requires group_id_x$1 == group_id_x$2 && group_id_y$1 ==
      group_id_y$2 && group_id_z$1 == group_id_z$2 ==>
      local_id_x$1 != local_id_x$2 || local_id_y$1 !=
      local_id_y$2 || local_id_z$1 != local_id_z$2;


implementation {:source_name "write_kernel"} {:kernel}
    $write_kernel($ni: bv32, $val: bv32, $nk: bv32)
{
  var {:source_name "out"} {:global} $$out: [bv32]bv32;
  var $n.0: bv32;
  var $i.0: bv32;
  var $idx.0$1: bv32;
  var $idx.0$2: bv32;
  var v0: bool;
  var v1: bool;
  var v2: bool;
  var _WRITE_READ_BENIGN_FLAG_$$out: bool;
  var _WRITE_READ_BENIGN_FLAG_$$in: bool;
  var _WRITE_HAS_OCCURRED_$$out: bool;

  assume !_READ_HAS_OCCURRED_$$in && !
      _WRITE_HAS_OCCURRED_$$in && !_ATOMIC_HAS_OCCURRED_$$in;
  assume !_READ_HAS_OCCURRED_$$out && !
      _WRITE_HAS_OCCURRED_$$out && !
      _ATOMIC_HAS_OCCURRED_$$out;

  v0 := $nk == 0bv32;

  if (!v0) {
    $n.0 := 0bv32;
```

```
assume {:captureState "loop_entry_state_0_0"} true;
while (BV32_ULT($n.0, $nk))
  invariant {:tag "accessedOffsetsSatisfyPredicates"}
      _b9 ==> _WRITE_HAS_OCCURRED_$$out ==> BV32_AND(
      BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
      num_groups_x), 4bv32), 1bv32), _WATCHED_OFFSET) ==
      BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
      num_groups_x), 4bv32), 1bv32), BV32_MUL(BV32_ADD(
      BV32_MUL(group_id_x$1, group_size_x), local_id_x$1)
      , 4bv32)) || BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(
      group_size_x, num_groups_x), 4bv32), 1bv32),
      _WATCHED_OFFSET) == BV32_AND(BV32_SUB(BV32_MUL(
      BV32_MUL(group_size_x, num_groups_x), 4bv32), 1bv32
      ), BV32_ADD(BV32_MUL(BV32_ADD(BV32_MUL(group_id_x$1
      , group_size_x), local_id_x$1), 4bv32), 1bv32)) ||
      BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
      num_groups_x), 4bv32), 1bv32), _WATCHED_OFFSET) ==
      BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
      num_groups_x), 4bv32), 1bv32), BV32_ADD(BV32_MUL(
      BV32_ADD(BV32_MUL(group_id_x$1, group_size_x),
      local_id_x$1), 4bv32), 2bv32)) || BV32_AND(BV32_SUB
      (BV32_MUL(BV32_MUL(group_size_x, num_groups_x), 4
      bv32), 1bv32), _WATCHED_OFFSET) == BV32_AND(
      BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
      num_groups_x), 4bv32), 1bv32), BV32_ADD(BV32_MUL(
      BV32_ADD(BV32_MUL(group_id_x$1, group_size_x),
      local_id_x$1), 4bv32), 3bv32));
  invariant {:tag "loopBound"} {:thread 1} _b3 ==>
      BV32_UGE($n.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b2 ==>
      BV32_ULE($n.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b1 ==>
      BV32_SGE($n.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b0 ==>
      BV32_SLE($n.0, 0bv32);
  invariant {:block_sourceloc} {:sourceloc_num 3} true;
{
  $i.0, $idx.0$1 := 0bv32, BV32_ADD(BV32_MUL(
      group_id_x$1, group_size_x), local_id_x$1);
  $idx.0$2 := BV32_ADD(BV32_MUL(group_id_x$2,
      group_size_x), local_id_x$2);
  assume {:captureState "loop_entry_state_1_0"} true;

  while (BV32_ULT($i.0, $ni))
    invariant {:tag "accessedOffsetsSatisfyPredicates"}
        _b10 ==> _WRITE_HAS_OCCURRED_$$out ==> BV32_AND(
        BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
        num_groups_x), 4bv32), 1bv32), _WATCHED_OFFSET)
        == BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(
```

```
      group_size_x, num_groups_x), 4bv32), 1bv32),
      BV32_MUL(BV32_ADD(BV32_MUL(group_id_x$1,
      group_size_x), local_id_x$1), 4bv32)) || BV32_AND
      (BV32_SUB(BV32_MUL(BV32_MUL(group_size_x,
      num_groups_x), 4bv32), 1bv32), _WATCHED_OFFSET)
      == BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(
      group_size_x, num_groups_x), 4bv32), 1bv32),
      BV32_ADD(BV32_MUL(BV32_ADD(BV32_MUL(group_id_x$1,
       group_size_x), local_id_x$1), 4bv32), 1bv32)) ||
       BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(group_size_x
      , num_groups_x), 4bv32), 1bv32), _WATCHED_OFFSET)
       == BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(
      group_size_x, num_groups_x), 4bv32), 1bv32),
      BV32_ADD(BV32_MUL(BV32_ADD(BV32_MUL(group_id_x$1,
       group_size_x), local_id_x$1), 4bv32), 2bv32)) ||
       BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(group_size_x
      , num_groups_x), 4bv32), 1bv32), _WATCHED_OFFSET)
       == BV32_AND(BV32_SUB(BV32_MUL(BV32_MUL(
      group_size_x, num_groups_x), 4bv32), 1bv32),
      BV32_ADD(BV32_MUL(BV32_ADD(BV32_MUL(group_id_x$1,
       group_size_x), local_id_x$1), 4bv32), 3bv32));
  invariant {:tag "loopBound"} {:thread 1} _b8 ==>
      BV32_UGE($i.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b7 ==>
      BV32_ULE($i.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b6 ==>
      BV32_SGE($i.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b5 ==>
      BV32_SLE($i.0, 0bv32);
  invariant {:tag "loopCounterIsStrided"} {:thread 1}
      _b4 ==> BV32_AND(BV32_SUB(BV32_MUL(group_size_x,
      num_groups_x), 1bv32), $idx.0$1) == BV32_AND(
      BV32_SUB(BV32_MUL(group_size_x, num_groups_x), 1
      bv32), BV32_ADD(BV32_MUL(group_id_x$1,
      group_size_x), local_id_x$1));
  invariant {:tag "loopCounterIsStrided"} {:thread 2}
      _b4 ==> BV32_AND(BV32_SUB(BV32_MUL(group_size_x,
      num_groups_x), 1bv32), $idx.0$2) == BV32_AND(
      BV32_SUB(BV32_MUL(group_size_x, num_groups_x), 1
      bv32), BV32_ADD(BV32_MUL(group_id_x$2,
      group_size_x), local_id_x$2));
  invariant {:block_sourceloc} {:sourceloc_num 5} true
      ;
{
  _WRITE_HAS_OCCURRED_$$out := (if true && _TRACKING
      && _WATCHED_OFFSET == BV32_MUL($idx.0$1, 4bv32)
      && _WATCHED_VALUE_$$out == $val then true else
      _WRITE_HAS_OCCURRED_$$out);
  _WRITE_READ_BENIGN_FLAG_$$out := (if true &&
```

```
    _TRACKING && _WATCHED_OFFSET == BV32_MUL($idx.0$1
    , 4bv32) && _WATCHED_VALUE_$$out == $val then
    $val != $$out[BV32_MUL($idx.0$1, 4bv32)] else
    _WRITE_READ_BENIGN_FLAG_$$out);

_WRITE_READ_BENIGN_FLAG_$$out := (if true &&
    _WRITE_HAS_OCCURRED_$$out && _WATCHED_OFFSET ==
    BV32_MUL($idx.0$2, 4bv32) then false else
    _WRITE_READ_BENIGN_FLAG_$$out);
assume {:do_not_predicate} {:check_id "check_state_0
    "} {:captureState "check_state_0"} {:sourceloc}
    {:sourceloc_num 7} true;

assume !(true && _WRITE_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_MUL($idx.0$2, 4bv32) &&
    _WATCHED_VALUE_$$out != $val);
assume !(true && _READ_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_MUL($idx.0$2, 4bv32) &&
    _WATCHED_VALUE_$$out != $val);
assume !(true && _ATOMIC_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_MUL($idx.0$2, 4bv32));

assume {:captureState "call_return_state_0"} {:
    procedureName "_CHECK_WRITE_$$out"} true;
$$out[BV32_MUL($idx.0$1, 4bv32)] := $val;
$$out[BV32_MUL($idx.0$2, 4bv32)] := $val;

_WRITE_HAS_OCCURRED_$$out := (if true && _TRACKING
    && _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$1,
     4bv32), 1bv32) && _WATCHED_VALUE_$$out == $val
    then true else _WRITE_HAS_OCCURRED_$$out);
_WRITE_READ_BENIGN_FLAG_$$out := (if true &&
    _TRACKING && _WATCHED_OFFSET == BV32_ADD(BV32_MUL
    ($idx.0$1, 4bv32), 1bv32) && _WATCHED_VALUE_$$out
     == $val then $val != $$out[BV32_ADD(BV32_MUL(
    $idx.0$1, 4bv32), 1bv32)] else
    _WRITE_READ_BENIGN_FLAG_$$out);

_WRITE_READ_BENIGN_FLAG_$$out := (if true &&
    _WRITE_HAS_OCCURRED_$$out && _WATCHED_OFFSET ==
    BV32_ADD(BV32_MUL($idx.0$2, 4bv32), 1bv32) then
    false else _WRITE_READ_BENIGN_FLAG_$$out);
assume {:do_not_predicate} {:check_id "check_state_1
    "} {:captureState "check_state_1"} {:sourceloc}
    {:sourceloc_num 8} true;

assume !(true && _WRITE_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
    bv32), 1bv32) && _WATCHED_VALUE_$$out != $val);
```

```
assume !(true && _READ_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
    bv32), 1bv32) && _WATCHED_VALUE_$$out != $val);
assume !(true && _ATOMIC_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
    bv32), 1bv32));

assume {:captureState "call_return_state_0"} {:
    procedureName "_CHECK_WRITE_$$out"} true;
$$out[BV32_ADD(BV32_MUL($idx.0$1, 4bv32), 1bv32)] :=
     $val;
$$out[BV32_ADD(BV32_MUL($idx.0$2, 4bv32), 1bv32)] :=
     $val;

_WRITE_HAS_OCCURRED_$$out := (if true && _TRACKING
    && _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$1,
     4bv32), 2bv32) && _WATCHED_VALUE_$$out == $val
    then true else _WRITE_HAS_OCCURRED_$$out);
_WRITE_READ_BENIGN_FLAG_$$out := (if true &&
    _TRACKING && _WATCHED_OFFSET == BV32_ADD(BV32_MUL
    ($idx.0$1, 4bv32), 2bv32) && _WATCHED_VALUE_$$out
     == $val then $val != $$out[BV32_ADD(BV32_MUL(
    $idx.0$1, 4bv32), 2bv32)] else
    _WRITE_READ_BENIGN_FLAG_$$out);

_WRITE_READ_BENIGN_FLAG_$$out := (if true &&
    _WRITE_HAS_OCCURRED_$$out && _WATCHED_OFFSET ==
    BV32_ADD(BV32_MUL($idx.0$2, 4bv32), 2bv32) then
    false else _WRITE_READ_BENIGN_FLAG_$$out);
assume {:do_not_predicate} {:check_id "check_state_2
    "} {:captureState "check_state_2"} {:sourceloc}
    {:sourceloc_num 9} true;

assume !(true && _WRITE_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
    bv32), 2bv32) && _WATCHED_VALUE_$$out != $val);
assume !(true && _READ_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
    bv32), 2bv32) && _WATCHED_VALUE_$$out != $val);
assume !(true && _ATOMIC_HAS_OCCURRED_$$out &&
    _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
    bv32), 2bv32));

assume {:captureState "call_return_state_0"} {:
    procedureName "_CHECK_WRITE_$$out"} true;
$$out[BV32_ADD(BV32_MUL($idx.0$1, 4bv32), 2bv32)] :=
     $val;
$$out[BV32_ADD(BV32_MUL($idx.0$2, 4bv32), 2bv32)] :=
     $val;
```

```
            _WRITE_HAS_OCCURRED_$$out := (if true && _TRACKING
                && _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$1,
                 4bv32), 3bv32) && _WATCHED_VALUE_$$out == $val
                then true else _WRITE_HAS_OCCURRED_$$out);
            _WRITE_READ_BENIGN_FLAG_$$out := (if true &&
                _TRACKING && _WATCHED_OFFSET == BV32_ADD(BV32_MUL
                ($idx.0$1, 4bv32), 3bv32) && _WATCHED_VALUE_$$out
                 == $val then $val != $$out[BV32_ADD(BV32_MUL(
                $idx.0$1, 4bv32), 3bv32)] else
                _WRITE_READ_BENIGN_FLAG_$$out);

            _WRITE_READ_BENIGN_FLAG_$$out := (if true &&
                _WRITE_HAS_OCCURRED_$$out && _WATCHED_OFFSET ==
                BV32_ADD(BV32_MUL($idx.0$2, 4bv32), 3bv32) then
                false else _WRITE_READ_BENIGN_FLAG_$$out);
            assume {:do_not_predicate} {:check_id "check_state_3
                "} {:captureState "check_state_3"} {:sourceloc}
                {:sourceloc_num 10} true;

            assume !(true && _WRITE_HAS_OCCURRED_$$out &&
                _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
                bv32), 3bv32) && _WATCHED_VALUE_$$out != $val);
            assume !(true && _READ_HAS_OCCURRED_$$out &&
                _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
                bv32), 3bv32) && _WATCHED_VALUE_$$out != $val);
            assume !(true && _ATOMIC_HAS_OCCURRED_$$out &&
                _WATCHED_OFFSET == BV32_ADD(BV32_MUL($idx.0$2, 4
                bv32), 3bv32));

            assume {:captureState "call_return_state_0"} {:
                procedureName "_CHECK_WRITE_$$out"} true;
            $$out[BV32_ADD(BV32_MUL($idx.0$1, 4bv32), 3bv32)] :=
                 $val;
            $$out[BV32_ADD(BV32_MUL($idx.0$2, 4bv32), 3bv32)] :=
                 $val;
            $i.0, $idx.0$1 := BV32_ADD($i.0, 1bv32), BV32_ADD(
                $idx.0$1, BV32_MUL(group_size_x, num_groups_x));
            $idx.0$2 := BV32_ADD($idx.0$2, BV32_MUL(group_size_x
                , num_groups_x));
            assume {:captureState "loop_back_edge_state_1_0"}
                true;
        }

        $n.0 := BV32_ADD($n.0, 1bv32);
        assume {:captureState "loop_back_edge_state_0_0"} true
            ;
    }
}
```

```
}

axiom (if group_size_y == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if group_size_z == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if num_groups_y == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if num_groups_z == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if group_size_x == 128bv32 then 1bv1 else 0bv1) != 0
    bv1;

axiom (if num_groups_x == 32768bv32 then 1bv1 else 0bv1) !=
    0bv1;

axiom (if global_offset_x == 0bv32 then 1bv1 else 0bv1) != 0
    bv1;

axiom (if global_offset_y == 0bv32 then 1bv1 else 0bv1) != 0
    bv1;

axiom (if global_offset_z == 0bv32 then 1bv1 else 0bv1) != 0
    bv1;

const {:local_id_y} local_id_y$1: bv32;

const {:local_id_y} local_id_y$2: bv32;

const {:local_id_z} local_id_z$1: bv32;

const {:local_id_z} local_id_z$2: bv32;

const {:group_id_y} group_id_y$1: bv32;

const {:group_id_y} group_id_y$2: bv32;

const {:group_id_z} group_id_z$1: bv32;

const {:group_id_z} group_id_z$2: bv32;

function {:bvbuiltin "bvsle"} BV32_SLE(bv32, bv32) : bool;

const {:existential true} _b0: bool;
```

```
function {:bvbuiltin "bvsge"} BV32_SGE(bv32, bv32) : bool;
const {:existential true} _b1: bool;
function {:bvbuiltin "bvule"} BV32_ULE(bv32, bv32) : bool;
const {:existential true} _b2: bool;
function {:bvbuiltin "bvuge"} BV32_UGE(bv32, bv32) : bool;
const {:existential true} _b3: bool;
function {:bvbuiltin "bvsub"} BV32_SUB(bv32, bv32) : bv32;
function {:bvbuiltin "bvand"} BV32_AND(bv32, bv32) : bv32;
const {:existential true} _b4: bool;
const {:existential true} _b5: bool;
const {:existential true} _b6: bool;
const {:existential true} _b7: bool;
const {:existential true} _b8: bool;
const _WATCHED_VALUE_$$in: bv32;

var _WRITE_READ_BENIGN_FLAG_$$in: bool;

const _WATCHED_VALUE_$$out: bv32;

var _TRACKING: bool;
function {:bvbuiltin "bvsgt"} BV32_SGT(bv32, bv32) : bool;
function {:bvbuiltin "bvslt"} BV32_SLT(bv32, bv32) : bool;
const {:existential true} _b9: bool;
const {:existential true} _b10: bool;
```

kernel_structured_nocalls.bpl

A.2

```
type _SIZE_T_TYPE = bv32;

procedure _ATOMIC_OP32(x: [bv32]bv32, y: bv32) returns (z$1:
    bv32, A$1: [bv32]bv32, z$2: bv32, A$2: [bv32]bv32);

var {:source_name "Histogram"} {:global} $$Histogram: [bv32]
   bv32;

axiom {:array_info "$$Histogram"} {:global} {:elem_width 32}
    {:source_name "Histogram"} {:source_elem_width 32} {:
   source_dimensions "*"} true;

var {:race_checking} {:global} {:elem_width 32} {:
   source_elem_width 32} {:source_dimensions "*"}
   _READ_HAS_OCCURRED_$$Histogram: bool;

var {:race_checking} {:global} {:elem_width 32} {:
   source_elem_width 32} {:source_dimensions "*"}
   _WRITE_HAS_OCCURRED_$$Histogram: bool;

var {:race_checking} {:global} {:elem_width 32} {:
   source_elem_width 32} {:source_dimensions "*"}
   _ATOMIC_HAS_OCCURRED_$$Histogram: bool;

const _WATCHED_OFFSET: bv32;

const {:global_offset_x} global_offset_x: bv32;

const {:global_offset_y} global_offset_y: bv32;

const {:global_offset_z} global_offset_z: bv32;

const {:group_id_x} group_id_x$1: bv32;

const {:group_id_x} group_id_x$2: bv32;

const {:group_size_x} group_size_x: bv32;

const {:group_size_y} group_size_y: bv32;

const {:group_size_z} group_size_z: bv32;

const {:local_id_x} local_id_x$1: bv32;

const {:local_id_x} local_id_x$2: bv32;

const {:num_groups_x} num_groups_x: bv32;
```

```
const {:num_groups_y} num_groups_y: bv32;

const {:num_groups_z} num_groups_z: bv32;

function {:bvbuiltin "bvadd"} BV32_ADD(bv32, bv32) : bv32;

function {:bvbuiltin "bvmul"} BV32_MUL(bv32, bv32) : bv32;

function {:bvbuiltin "bvult"} BV32_ULT(bv32, bv32) : bool;

procedure {:source_name "reduceKernel"} {:kernel}
   $reduceKernel($nSubHists: bv32);
  requires !_WRITE_HAS_OCCURRED_$$Histogram && !
     _ATOMIC_HAS_OCCURRED_$$Histogram;
  requires BV32_SGT(group_size_x, 0bv32);
  requires BV32_SGT(num_groups_x, 0bv32);
  requires BV32_SGE(group_id_x$1, 0bv32);
  requires BV32_SGE(group_id_x$2, 0bv32);
  requires BV32_SLT(group_id_x$1, num_groups_x);
  requires BV32_SLT(group_id_x$2, num_groups_x);
  requires BV32_SGE(local_id_x$1, 0bv32);
  requires BV32_SGE(local_id_x$2, 0bv32);
  requires BV32_SLT(local_id_x$1, group_size_x);
  requires BV32_SLT(local_id_x$2, group_size_x);
  requires BV32_SGT(group_size_y, 0bv32);
  requires BV32_SGT(num_groups_y, 0bv32);
  requires BV32_SGE(group_id_y$1, 0bv32);
  requires BV32_SGE(group_id_y$2, 0bv32);
  requires BV32_SLT(group_id_y$1, num_groups_y);
  requires BV32_SLT(group_id_y$2, num_groups_y);
  requires BV32_SGE(local_id_y$1, 0bv32);
  requires BV32_SGE(local_id_y$2, 0bv32);
  requires BV32_SLT(local_id_y$1, group_size_y);
  requires BV32_SLT(local_id_y$2, group_size_y);
  requires BV32_SGT(group_size_z, 0bv32);
  requires BV32_SGT(num_groups_z, 0bv32);
  requires BV32_SGE(group_id_z$1, 0bv32);
  requires BV32_SGE(group_id_z$2, 0bv32);
  requires BV32_SLT(group_id_z$1, num_groups_z);
  requires BV32_SLT(group_id_z$2, num_groups_z);
  requires BV32_SGE(local_id_z$1, 0bv32);
  requires BV32_SGE(local_id_z$2, 0bv32);
  requires BV32_SLT(local_id_z$1, group_size_z);
  requires BV32_SLT(local_id_z$2, group_size_z);
  requires group_id_x$1 == group_id_x$2 && group_id_y$1 ==
     group_id_y$2 && group_id_z$1 == group_id_z$2 ==>
     local_id_x$1 != local_id_x$2 || local_id_y$1 !=
     local_id_y$2 || local_id_z$1 != local_id_z$2;
```

```
  modifies $$Histogram, _WRITE_HAS_OCCURRED_$$Histogram,
      _WRITE_READ_BENIGN_FLAG_$$Histogram,
      _WRITE_READ_BENIGN_FLAG_$$Histogram;

implementation {:source_name "reduceKernel"} {:kernel}
    $reduceKernel($nSubHists: bv32)
{
  var $bin.0$1: bv32;
  var $bin.0$2: bv32;
  var $i.0: bv32;
  var v0$1: bv32;
  var v0$2: bv32;
  var v1: bool;
  var v2$1: bv32;
  var v2$2: bv32;

  var _READ_HAS_OCCURRED_$$Histogram: bool;
  assume !_READ_HAS_OCCURRED_$$Histogram;

  v0$1 := BV32_ADD(BV32_MUL(group_id_x$1, group_size_x),
     local_id_x$1);
  v0$2 := BV32_ADD(BV32_MUL(group_id_x$2, group_size_x),
     local_id_x$2);
  $bin.0$1, $i.0 := 0bv32, 0bv32;
  $bin.0$2 := 0bv32;
  assume {:captureState "loop_entry_state_0_0"} true;

  while ( BV32_ULT($i.0, $nSubHists) )
    invariant {:tag "accessBreak"} _b7 ==>
       _READ_HAS_OCCURRED_$$Histogram ==> group_id_x$1 ==
       BV32_DIV(_WATCHED_OFFSET, group_size_x);
    invariant {:tag "accessedOffsetsSatisfyPredicates"} _b6
       ==> _READ_HAS_OCCURRED_$$Histogram ==> BV32_AND(
       BV32_SUB(BV32_MUL(1bv32, 256bv32), 1bv32),
       _WATCHED_OFFSET) == BV32_AND(BV32_SUB(BV32_MUL(1bv32,
        256bv32), 1bv32), BV32_ADD(BV32_MUL(0bv32, 256bv32),
        BV32_ADD(BV32_MUL(group_id_x$1, group_size_x),
       local_id_x$1)));
    invariant {:tag "accessUpperBoundBlock"} _b5 ==>
       _READ_HAS_OCCURRED_$$Histogram ==> BV32_SLT(
       _WATCHED_OFFSET, BV32_ADD(BV32_MUL(BV32_ADD(
       group_id_x$1, 1bv32), group_size_x), local_id_x$1));
    invariant {:tag "accessLowerBoundBlock"} _b4 ==>
       _READ_HAS_OCCURRED_$$Histogram ==> BV32_SLE(BV32_ADD(
       BV32_MUL(group_id_x$1, group_size_x), local_id_x$1),
       _WATCHED_OFFSET);
    invariant {:tag "loopBound"} {:thread 1} _b3 ==>
       BV32_UGE($i.0, 0bv32);
    invariant {:tag "loopBound"} {:thread 1} _b2 ==>
```

```
      BV32_ULE($i.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b1 ==>
      BV32_SGE($i.0, 0bv32);
  invariant {:tag "loopBound"} {:thread 1} _b0 ==>
      BV32_SLE($i.0, 0bv32);
  invariant {:block_sourceloc} {:sourceloc_num 1} true;
{
  assume {:partition} v1;
  _READ_HAS_OCCURRED_$$Histogram := (if true && _TRACKING
      && _WATCHED_OFFSET == BV32_ADD(BV32_MUL($i.0, 256bv32
      ), v0$1) && _WATCHED_VALUE_$$Histogram == $$Histogram
      [BV32_ADD(BV32_MUL($i.0, 256bv32), v0$1)] then true
      else _READ_HAS_OCCURRED_$$Histogram);
  assume {:do_not_predicate} {:check_id "check_state_1"}
      {:captureState "check_state_1"} {:sourceloc} {:
      sourceloc_num 3} true;
  assume !(true && _WRITE_HAS_OCCURRED_$$Histogram &&
      _WATCHED_OFFSET == BV32_ADD(BV32_MUL($i.0, 256bv32),
      v0$2) && _WRITE_READ_BENIGN_FLAG_$$Histogram);
  assume !(true && _ATOMIC_HAS_OCCURRED_$$Histogram &&
      _WATCHED_OFFSET == $$Histogram[BV32_ADD(BV32_MUL($i
      .0, 256bv32), v0$2)]);
  assume {:captureState "call_return_state_0"} {:
      procedureName "_CHECK_READ_$$Histogram"} true;
  v2$1 := $$Histogram[BV32_ADD(BV32_MUL($i.0, 256bv32),
      v0$1)];
  v2$2 := $$Histogram[BV32_ADD(BV32_MUL($i.0, 256bv32),
      v0$2)];
  $bin.0$1, $i.0 := BV32_ADD($bin.0$1, v2$1), BV32_ADD($i
      .0, 1bv32);
  $bin.0$2 := BV32_ADD($bin.0$2, v2$2);
  assume {:captureState "loop_back_edge_state_0_0"} true;
}

assume {:partition} !v1;
_WRITE_HAS_OCCURRED_$$Histogram := (if true && _TRACKING
    && _WATCHED_OFFSET == v0$1 &&
    _WATCHED_VALUE_$$Histogram == $bin.0$1 then true else
    _WRITE_HAS_OCCURRED_$$Histogram);
_WRITE_READ_BENIGN_FLAG_$$Histogram := (if true &&
    _TRACKING && _WATCHED_OFFSET == v0$1 &&
    _WATCHED_VALUE_$$Histogram == $bin.0$1 then $bin.0$1 !=
     $$Histogram[v0$1] else
    _WRITE_READ_BENIGN_FLAG_$$Histogram);
_WRITE_READ_BENIGN_FLAG_$$Histogram := (if true &&
    _WRITE_HAS_OCCURRED_$$Histogram && _WATCHED_OFFSET ==
    v0$2 then false else
    _WRITE_READ_BENIGN_FLAG_$$Histogram);
assume {:do_not_predicate} {:check_id "check_state_0"} {:
```

```
      captureState "check_state_0"} {:sourceloc} {:
      sourceloc_num 6} true;
  assume !(true && _WRITE_HAS_OCCURRED_$$Histogram &&
      _WATCHED_OFFSET == v0$2 && _WATCHED_VALUE_$$Histogram
      != $bin.0$2);
  assume !(true && _READ_HAS_OCCURRED_$$Histogram &&
      _WATCHED_OFFSET == v0$2 && _WATCHED_VALUE_$$Histogram
      != $bin.0$2);
  assume !(true && _ATOMIC_HAS_OCCURRED_$$Histogram &&
      _WATCHED_OFFSET == v0$2);
  assume {:captureState "call_return_state_0"} {:
      procedureName "_CHECK_WRITE_$$Histogram"} true;
  $$Histogram[v0$1] := $bin.0$1;
  $$Histogram[v0$2] := $bin.0$2;
  return;
}


axiom (if group_size_y == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if group_size_z == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if num_groups_y == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if num_groups_z == 1bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if group_size_x == 128bv32 then 1bv1 else 0bv1) != 0
    bv1;

axiom (if num_groups_x == 2bv32 then 1bv1 else 0bv1) != 0bv1
    ;

axiom (if global_offset_x == 0bv32 then 1bv1 else 0bv1) != 0
    bv1;

axiom (if global_offset_y == 0bv32 then 1bv1 else 0bv1) != 0
    bv1;

axiom (if global_offset_z == 0bv32 then 1bv1 else 0bv1) != 0
    bv1;

const {:local_id_y} local_id_y$1: bv32;

const {:local_id_y} local_id_y$2: bv32;
```

```
const {:local_id_z} local_id_z$1: bv32;

const {:local_id_z} local_id_z$2: bv32;

const {:group_id_y} group_id_y$1: bv32;

const {:group_id_y} group_id_y$2: bv32;

const {:group_id_z} group_id_z$1: bv32;

const {:group_id_z} group_id_z$2: bv32;

function {:bvbuiltin "bvsle"} BV32_SLE(bv32, bv32) : bool;

const {:existential true} _b0: bool;

function {:bvbuiltin "bvsge"} BV32_SGE(bv32, bv32) : bool;

const {:existential true} _b1: bool;

function {:bvbuiltin "bvule"} BV32_ULE(bv32, bv32) : bool;

const {:existential true} _b2: bool;

function {:bvbuiltin "bvuge"} BV32_UGE(bv32, bv32) : bool;

const {:existential true} _b3: bool;

const _WATCHED_VALUE_$$Histogram: bv32;

var _WRITE_READ_BENIGN_FLAG_$$Histogram: bool;

procedure {:inline 1} _LOG_ATOMIC_$$Histogram(_P: bool,
    _offset: bv32);
modifies _ATOMIC_HAS_OCCURRED_$$Histogram;

implementation {:inline 1} _LOG_ATOMIC_$$Histogram(_P: bool,
     _offset: bv32)
{
log_access_entry:
_ATOMIC_HAS_OCCURRED_$$Histogram := (if _P && _TRACKING &&
    _WATCHED_OFFSET == _offset then true else
    _ATOMIC_HAS_OCCURRED_$$Histogram);
return;
}

procedure _CHECK_ATOMIC_$$Histogram(_P: bool, _offset: bv32)
    ;
requires {:source_name "Histogram"} {:array "$$Histogram"}
```

```
    {:race} {:write_atomic} !(_P &&
    _WRITE_HAS_OCCURRED_$$Histogram && _WATCHED_OFFSET ==
    _offset);
requires {:source_name "Histogram"} {:array "$$Histogram"}
    {:race} {:read_atomic} !(_P &&
    _READ_HAS_OCCURRED_$$Histogram && _WATCHED_OFFSET ==
    _offset);

var _TRACKING: bool;

function {:bvbuiltin "bvsgt"} BV32_SGT(bv32, bv32) : bool;

function {:bvbuiltin "bvslt"} BV32_SLT(bv32, bv32) : bool;

const {:existential true} _b4: bool;

const {:existential true} _b5: bool;

function {:bvbuiltin "bvsub"} BV32_SUB(bv32, bv32) : bv32;

function {:bvbuiltin "bvand"} BV32_AND(bv32, bv32) : bv32;

const {:existential true} _b6: bool;

function {:bvbuiltin "bvsdiv"} BV32_DIV(bv32, bv32) : bv32;

const {:existential true} _b7: bool;
```

HistogramAtomics.bpl

A.3

```
const {:existential true} _b1: bool;
const {:existential true} _b2: bool;
const {:existential true} _b3: bool;
const {:existential true} _b4: bool;
const {:existential true} _b5: bool;
const {:existential true} _b6: bool;
const {:existential true} _b7: bool;
const {:existential true} _b8: bool;
const {:existential true} _b9: bool;

function lol(b: bool): bool;

procedure foo (m: int) returns (res: int)
  requires m >= 0;
  ensures res == m;
{
```

```
var i: int;
var j: int;
var stop: bool;

i := 0;
j := 0;
stop := false;

res := 0;

while (!stop)
  invariant _b1 ==> i <= 0;
  invariant _b2 ==> i <= 1;
  invariant _b3 ==> i <= 2;
  invariant _b4 ==> i <= 3;
  invariant _b5 ==> i <= 4;
  invariant _b6 ==> i <= 5;
  invariant _b7 ==> i <= 6;
  invariant _b8 ==> i <= 7;
  invariant _b9 ==> i <= 8;
{
  i := i + 1;
  stop := lol(stop);
}

// the next if statement is repeated 50000 times

if (res != m)
{
  j := j + 1;
}
else
{
  j := j - 1;
}

      ⋮

if (res != m)
{
  j := j + 1;
}
else
{
  j := j - 1;
}

res := m;
```

```
}
```

many_false_invariants.bpl

A.4

```
const {:existential true} _b1: bool;
const {:existential true} _b2: bool;

function lol(b: bool): bool;

procedure foo (m: int) returns (res: int)
  requires m >= 0;
  ensures res == m;
{
  var i: int;
  var j: int;
  var stop: bool;

  i := 0;
  j := 0;
  stop := false;

  res := 0;

  while (!stop)
    invariant _b1 ==> i <= 0;
    invariant _b2 ==> i <= 1;
  {
    i := i + 1;
    stop := lol(stop);
  }

  // the next if statement is repeated 50000 times

  if (res != m)
  {
    j := j + 1;
  }
  else
  {
    j := j - 1;
  }

        ⋮

  if (res != m)
```

```
  {
    j := j + 1;
  }
  else
  {
    j := j - 1;
  }

  res := m;

}
```

many_lines.bpl

A.5

```
const {:existential true} _b1: bool;
const {:existential true} _b2: bool;
const {:existential true} _b3: bool;

procedure multiply_by_4 (m: int) returns (res: int)
  requires m > 0;
  ensures res == 4 * m;
{
  var i: int;
  var j: int;
  i := 0;
  j := 0;
  res := 0;

  while (i < 4)
    invariant _b1 ==> res == m * i;
    invariant _b2 ==> i <= 4;
    invariant j == 0;
  {
    while (j < m)
      invariant _b3 ==> res == (m * i) + j;
      invariant j <= m;
    {
      j := j + 1;
      res := res + 1;
    }

    i := i + 1;
    j := 0;
  }
}
```

nested_loops_transformed_09.bpl

A.6

```
const {:existential true} _b1: bool;
const {:existential true} _b2: bool;
const {:existential true} _b3: bool;
const {:existential true} _b4: bool;

function lol(b: bool): bool;

procedure bar() returns ()
{
  var a: int;
  var b: int;
  var c: int;
  var temp: int;
  var stop : bool;
  stop := false;
  a := 1;
  b := 2;
  c := 3;
  while (!stop)
    invariant _b3 ==> a != c;
    invariant _b1 ==> a != b;
    invariant _b2 ==> b != c;
    invariant _b4 ==> c > 1;
  {
    stop := lol(stop);
    temp := a;
    a := b;
    b := c;
    c := temp;
  }

  assert {:msg "outer"} a != c; // should fail without _b2
      candidate, should succeed otherwise.
}
```

unsound_00.bpl

Bibliography

[1] AMD. Accelerated parallel processing sdk. http://developer.amd.com/sdks/amdappsdk. [Online; accessed 2014].

[2] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for GPU kernels. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 113–132. ACM, 2012.

[3] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[4] Cormac Flanagan, Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[5] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[6] K. Rustan M. Leino. This is boogie 2. https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/, June 2008.

[7] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 41–62, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[8] Tobias Nipkow and Gerwin Klein. *IMP: A Simple Imperative Language*, pages 75–94. Springer International Publishing, Cham, 2014.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Lightweight automatic loop invariant selection

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Pozdnyakov	Pavel

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 30. September 2019	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.