A parallelizable approach for mining likely invariants

Alessandro Danese Luca Piccolboni Graziano Pravadelli
University of Verona, Strada le Grazie 15
37134, Verona, Italy
name.surname@univr.it

ABSTRACT

A relevant aspect in design analysis and verification is monitoring how logic relations among different variables change at run time. Current static approaches suffer from scalability problems that prevent their adoption on large designs. On the contrary, dynamic techniques scale better from the memory-consumption point of view. However, to achieve a high accuracy, they require to analyse a huge number of (long) execution traces, which results in time-consuming phases. In this paper, we present a new efficient approach to automatically infer logic relations among the variables of a design implementation. Both a sequential and a GPU-oriented parallel implementation are proposed to dynamically extract likely invariants from execution traces on different time windows. Execution traces composed of millions of simulation instants can be efficiently analysed.

1. INTRODUCTION

Automatic invariant detection is a widely adopted strategy to analyse several aspects in verification of both SW programs and HW designs. Without forgetting the importance of invariants for documentation purposes, invariant inference has been used, for example, for test generation [5], analysis of dynamic memory consumption [4], static checking [13], detection of race conditions [14], identification of memory access violations [10], generic bug catching [15], and, more recently, also mining of temporal assertions [6]. Independently from its use, an invariant is a logic formula that holds between a couple (or several couples) of points, A and B, of an implementation, thus expressing a stable condition in the behaviour of the system under verification (SUV) for all its executions. Possibly, A and B may correspond, respectively, to the beginning and the end of the SUV execution. Different kinds of invariants, like, for example $(x \leq y)$, $(ax + b = y)$, $(x \neq NULL)$, can be inferred by either static or dynamic analysis of the SUV.

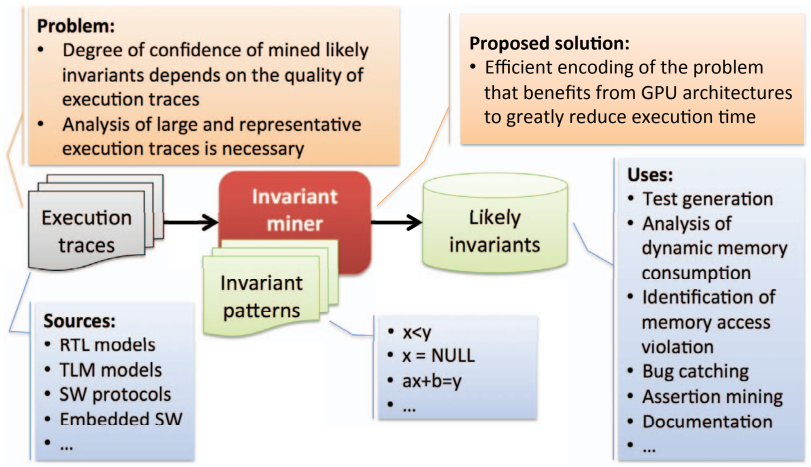
Static approaches, like [8, 16], are exhaustive and work well for relatively small/medium-size implementations. However, they generally require to formally analyse the source code and they scale badly for large SUVs.

An alternative to static approaches is represented by dynamic invariant mining [7]. In this case, invariants are extracted by analysing a finite set of execution traces obtained from the simulation of the SUV. Dynamic inference works even if the source code is not available and it scales better for large SUVs. Indeed, the efficiency of dynamic miners is more related to the length of analysed execution traces and the number of observed variables than the complexity of the SUV.

As a drawback, these techniques, being not exhaustive, can extract only *likely invariants*, i.e., properties that are only statistically true during the simulation of the SUV. Then, to increase the degree of confidence on invariants mined by dynamic approaches, a large (and representative) set of execution traces must be analysed. However, for complex HW designs this could require to analyse thousands of execution traces, including millions of clock cycles, and predicating over hundreds of variables, which becomes an unmanageable time-consuming activity for existing approaches. Similar considerations apply also for execution traces derived from embedded SW applications.

To overcome this limitation, this paper proposes a new approach for dynamically mining likely invariants (Fig. 1) that greatly reduces the execution time with respect to existing techniques, without affecting the accuracy of the analysis. Moreover, thanks to an efficient encoding of information required for the mining procedure, a parallel version of the proposed approach is also presented that benefits from execution on graphics processing unit (GPU) architectures. Both the sequential and the parallel versions allow to manage the mining on execution traces composed of millions of simulation instants in few seconds. The two algorithms can work on execution traces from both HW and SW domains. Moreover, when an exhaustive mining of invariants on different time windows, belonging to the same execution trace, is required (for example, for extracting invariants to be used in mining of temporal assertions [6]), the parallel version can analyse hundreds of thousands of sub-traces on the order of minutes.

The rest of the paper is organized as follows. Section 2 summarizes the state of the art. Section 3 reports preliminary concepts and definitions. Section 4 presents the proposed



Problem:
Degree of confidence of mined likely invariants depends on the quality of execution traces
Analysis of large and representative execution traces is necessary
Proposed solution:
Efficient encoding of the problem that benefits from GPU architectures to greatly reduce execution time
Execution traces
Invariant miner
Likely invariants
Invariant patterns
Uses:
Test generation
Analysis of dynamic memory consumption
Identification of memory access violation
Bug catching
Assertion mining
Documentation
...
Sources:
RTL models
TLM models
SW protocols
Embedded SW
...
x<y
x = NULL
ax+b=y
...

Figure 1: Dynamic mining of likely invariants.

mining approach. Finally, Section 5 and Section 6 deal with experimental results and concluding remarks.

2. RELATED WORK

To the best of our knowledge the most effective and flexible miner of likely invariants is Daikon [7]. It analyses execution traces through an inference engine that incrementally detects invariants according to templates specified in a grammar configuration file. To extract invariants on specific points of a program, code instrumentation is required. Daikon has been mainly used for documentation, debugging, testing and maintainability of SW programs. A brief overview of Daikon's functionalities is reported in Section 3.2.

Several mining tools alternative to Daikon, and the relative uses, are referenced in [2]. In this section, we refer to some of them in representation of different categories (commercial vs. academic, hardware- vs. software-oriented). For example, from a commercial point of view, Daikon inspired the creation of Agitator [11] that dynamically extracts invariants to check their compliance with respect to manually defined conditions. An alternative academic approach is implemented in DIDUCE [15]. It aids programmers to identify root causes of errors on Java programs. DIDUCE's engine dynamically formulates strict invariant hypotheses obeyed by the program at the beginning, then it gradually relaxes such hypotheses when violations are detected to include new behaviours. Finally, in the HW domain, IODINE [9] infers likely invariants for HW design descriptions. Inferred invariants refer to state-machine protocols, request-acknowledge pairs, and mutual exclusion between signals.

Contrary to the approach proposed in the current paper, previous approaches require the instrumentation of program points which can be done only when the source code of the SUV is available. Moreover, they cannot take advantage of massive parallel execution on GPUs, thus they scale badly for large sets of long execution traces.

3. BACKGROUND

This section first reports preliminary definitions that are necessary to understand the proposed methodology. Then it summarizes the main idea underlying the Daikon miner, which is used as a comparison in the experimental results. Finally, it presents a brief overview of the GPU architecture to create the necessary background for describing the parallel version of the proposed mining approach.

3.1 Preliminary definitions

In the context of dynamic mining of likely invariants the following definitions are relevant.

Definition 1. Given a finite sequence of simulation instants $\langle t_1, \ldots, t_n \rangle$ and the set of variables V of a model $\mathcal{M}$, an **execution trace** of $\mathcal{M}$ is a finite sequence of pairs $T = \langle (V_1, t_1), \ldots, (V_n, t_n) \rangle$ where $V_i = eval(V, t_i)$ is the evaluation of variables in V at simulation instant t_i.

Definition 2. Given an execution trace $T = \langle (V_1, t_1), \ldots, (V_n, t_n) \rangle$, and two simulation instants t_i and t_j such that $1 \leq t_i \leq t_j \leq n$, a **time window** $TW_{i,j} = \langle (V_i, t_i), \ldots, (V_j, t_j) \rangle$ is a subsequence of contiguous elements of T.

More informally, an execution trace describes for each simulation instant t_i the values assumed by each variable included in V during the evolution of the model $\mathcal{M}$. A time window is a sub-trace of an execution trace. Our approach automatically infers likely invariants by analysing a whole execution trace as well as a sequence of its time windows.

Definition 3. Given a set of variables V of a model $\mathcal{M}$ and an execution trace T, a **trace invariant** (T-invariant) is a logic formula over V that is true for each simulation instant in T.

Definition 4. Given a set of variables V of a model $\mathcal{M}$, an execution trace T, and a time window $TW_{i,j} \subseteq T$, a **time window invariant** (TW-invariant) is a logic formula over V that is true for each simulation instant in $TW_{i,j}$.

3.2 Daikon

Daikon analyses the execution traces through an inference engine that incrementally detects likely invariants according to a list of templates specified in a configuration file. The execution traces are generally obtained by running an instrumented target program that reports the values of several program points. Usually, the most used program points on which Daikon infers invariants are global variables and input/output arguments of methods. The internal engine of Daikon can be represented as a hierarchy of classes. Each of them implements a checker for a specific arithmetic/logic pattern between variables. Several variables' domains are currently supported, e.g., Daikon can extract likely invariants for Boolean, numeric, string and vector variables. The main idea behind the incremental invariant-inference engine of Daikon can be summarized in three steps: 1) instantiate a candidate invariant (i.e., a class) for each selected template given a combination of variables; 2) remove the candidate invariants contradicted by a sample of the trace; and 3) report the invariants that remain after processing all the samples, and after applying post-processing filtering. In order to efficiently extract invariants many optimizations have been implemented in Daikon. The most relevant of them are:

- If two or more variables are always equal, then any invariant that can be verified for one of those variables is also verified for each of the other variables.
- A dynamically constant variable is one that has the same value at each observed sample. The invariant $x = a$ (for constant a) makes any other invariant over (only) x redundant.
- Suppression of invariants logically implied by some set of other invariants is adopted. For example, $x > y$ implies $x \geq y$, and $0 < x < y$ and $z = 0$ imply $x div y = z$.

With respect to the approach proposed in Daikon, in this paper we do not need code instrumentation, and we encode information on candidate invariants by means of a vector-based data structure, which is more efficient and particularly suited for parallel computing, as proved by experimental results. On the contrary, the inference engine of Daikon cannot be easily ported on a GPU. To the best of our knowledge this is the first implementation of an invariant miner that runs on a GPU.

3.3 GPU architecture

GPUs are multi-core coprocessors originally intended to speed-up computer graphics. However, their highly-parallel structure makes GPUs powerful devices also for the elaboration of general-purpose computing-intensive processes that work in parallel on large blocks of data. This approach is commonly known as general-purpose computing on graphics processing units (GPGPU). The affirmation of GPGPU was further supported by the definition of ad hoc parallel computing platforms and programming models, like CUDA [1] and OpenCL [3]. Figure 2 shows the internal architecture of common GPUs. A GPU is composed of various (streaming) multiprocessors, each one consisting of several processing cores that execute in parallel a sequence of instructions, commonly known as *kernel-function*. Multiple program threads

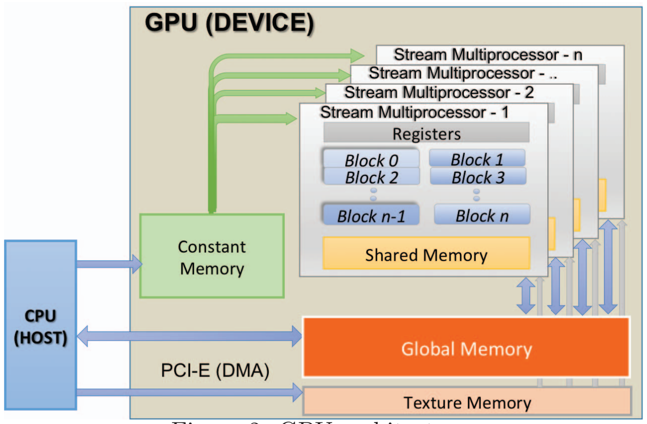


Figure 2: GPU architecture.

organized in *blocks* are distributed and concurrently executed by the cores of each multiprocessor. Inside a multiprocessor, data are elaborated in SIMD (single instruction, multiple data) mode. As a consequence, threads running on the same core that need to perform instructions on different branches of a conditional statement are executed sequentially. This issue is known as "divergence" and it is one of the most important cause of performance degradation in GPGPU. In this computational platform, there are four types of memories, namely *shared*, *constant*, *texture* and *global* memory. All of them, but shared memory, are freely accessible by an external CPU, which is used to submit kernels to the GPU. The shared memory is very fast and it is available only for threads belonging to the same block for data sharing.

4. INVARIANT MINING

This paper presents a dynamic parallelizable approach for mining both trace invariants and time window invariants. Indeed, a T-invariant is a TW-invariant for a time window that extends from the first to the last simulation instant of the corresponding trace. Thus, to avoid burdening the discussion, in the following, we use the term invariant when concepts apply indistinctly to T-invariants and TW-invariants. We first propose a simple but efficient sequential algorithm (Section 4.1) to dynamically infer invariants. Then, changes we made to implement a faster parallel version running on a GPU are discussed (Section 4.2).

Given a set of variables V of a model $\mathcal{M}$, both the sequential and parallel algorithms rely on a bit vector-based data structure, called *invariant vector*, to efficiently represent logic relations among the variables in V. Without lack of generality, let us consider a time window TW and a list of n invariant templates $I = \{inv_1, inv_2, \ldots, inv_n\}$ representing logic relations among the set of variables V. We track if a m-ary logic relation corresponding to the invariant $inv_i \in I$, instantiated with a tuple[1] of variables $(v_1, \ldots, v_m) \in V^m$, holds in TW by using an invariant vector, inv_result, composed of n elements. Element i of inv_result corresponds to the instance $inv_i(v_1, \ldots, v_m)$ of the invariant template inv_i referred to the tuple $(v_1, \ldots, v_m)$. Thus, $inv\_result[i]$ is 0 if $inv_i(v_1, \ldots, v_m)$ is false at least once in TW; it is 1 otherwise.

[1]The arity of the tuple depends on the arity of the invariant.

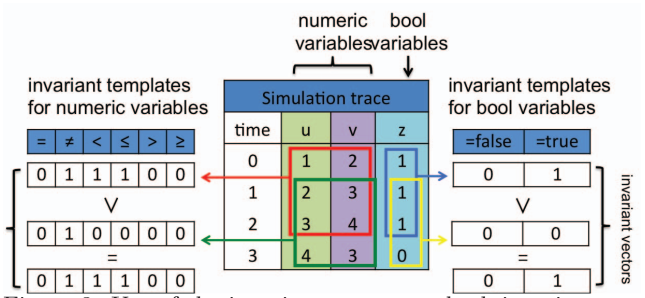


Figure 3: Use of the invariant vector to check invariants on different time windows.

Given a set of execution traces and a set of different time windows, this invariant vector allows us to rapidly analyse the following conditions, for all instances of the invariant templates, as described in Sections 4.1 and 4.2:

[C1] $inv_i(v_1, \ldots, v_m)$ is true for at least one time window of one execution trace. This is necessary, for example, to prove that there exists at least one execution run that brings the model to a stable condition where $inv_i(v_1, \ldots, v_m)$ remains true for a given time interval.

[C2] $inv_i(v_1, \ldots, v_m)$ is true for at least one time window of all the considered execution traces. This shows a stable condition occurs, where $inv_i(v_1, \ldots, v_m)$ is true, for a given time interval at least once per each execution run of the model.

[C3] $inv_i(v_1, \ldots, v_m)$ is true for at least one execution trace. This can prove that there exist at least one run of the model where the condition $inv_i(v_1, \ldots, v_m)$ remains always stable for the entire duration of the execution run.

[C4] $inv_i(v_1, \ldots, v_m)$ is true for all the analysed execution traces. This statistically proves $inv_i(v_1, \ldots, v_m)$ holds always each time the model is executed, assuming that the analysed traces are statistically representative of all the model's behaviours.

For example, in Figure 3, the use of the invariant vector is reported for a simple execution trace involving two numeric variables (u and v) and one Boolean variable (z). Two time windows of length 3 are highlighted, related, respectively, to the time intervals [0,2] and [1,3]. The six logic relations on the left and the two on the right are used as invariant templates, respectively, for the numeric and the Boolean variables. By considering only numeric variables (same considerations apply for the Boolean variable), in the first time window, the invariant templates $u \neq v$ and $u < v$ are true (red box), thus the corresponding invariant vector is $\{0, 1, 1, 1, 0, 0\}$. Meanwhile, in the second time window only the invariant template $u \neq v$ is true (green box), thus the corresponding invariant vector is $\{0, 1, 0, 0, 0, 0\}$. As a consequence, a global invariant vector for the numeric variables, for example to check condition C1, is obtained by applying a bitwise OR among the invariant vectors of each time window. Condition C2 is checked by a bitwise AND among the global invariant vectors of different execution traces. Finally, C3 and C4 are similarly obtained by analysing the whole execution traces without time-window partitioning.

4.1 Sequential algorithm

In the current implementation, our algorithm can infer binary logic relations represented by the following invariant templates

- $\{(u = v), (u \neq v), (u < v), (u \leq v), (u > v), (u \geq v)\}$ for a pair of numeric variables (u, v);
- $\{(v = true), (v = false)\}$ for a Boolean variable v.

However, the approach is independent from the specific template, thus it can be easily extended to consider further kinds of arithmetic logic relations between two or more variables and constants, like, for example, being in a range $(a \leq v \leq b)$ and linear relationships $(v = au + b)$.

The sequential approach follows the strategy implemented in Algorithm 1. Given a set V of variables, an execution trace T and an integer $l > 0$, it extracts all formulas that hold on at least one time window of length l included in T. This paper is intended to present the mining algorithm and its optimization for parallel computing, while no consideration is reported on the choice of the length of the time windows. Indeed, the selection of the value for the parameter l depends on the desired kind of verification. For example, by varying the parameter l, different time window intervals can be analysed to check conditions of kind C1. On the contrary, if l is set to the size of T, the algorithm computes invariants holding on the whole execution trace, thus providing results for analysing conditions of kind C3. Finally, calling the algorithm on several execution traces, the existence of invariants satisfying conditions C2 and C4 can be analysed too.

Assuming the presence of two sets of invariant templates: I_{Bool} for Boolean variables and I_{Num} for numeric variables, the algorithm starts by invoking the function *invariantChecker*, which calls *getBoolInv* and *getNumInv*, respectively, on each Boolean variable $u \in V$ and on each pair of numeric variables $(u, v) \in V \times V$.

The execution flow of *getBoolInv* and *getNumInv* is practically the same. They first initialize elements of the invariant vector *inv_result* to 0 (lines 17 and 37). In the current implementation, we have 6 invariant templates for numeric variables and 2 for Boolean variables, as described at the beginning of this section. At the end of the algorithm execution, *inv_result*$[i]$ is 1 if the corresponding invariant inv_i holds at least on one time window of T. Then, *getBoolInv* and *getNumInv* iterate the following steps for each time window of length l belonging to the execution trace T (lines 18-32 and 38-54):

1. Before starting the analysis of a new time window, another invariant vector (*local_res*) of the same length of *inv_result* is initialized to 1 (lines 19 and 39). At the end of the time window analysis, *local_res*$[i]$ is 1 if no counterexample has been found for the corresponding invariant inv_i.
2. During the analysis of a time window, *local_res*$[i]$ is set to 0 as soon as a counter example is found within

Algorithm 1 invariant_checker - Sequential Algorithm

```
 1: function INVARIANT_CHECKER($T, l, V$)
 2:     for all $u \in V$ do
 3:         if getType($u$) == BOOL then
 4:             print(GETBOOLINV($T, l, u$));
 5:         end if
 6:         if getType($u$) == NUMERIC then
 7:             for all $v \in V \wedge u \neq v$ do
 8:                 if getType($v$) == NUMERIC then
 9:                     print(GETNUMINV($T, l, u, v$));
10:                 end if
11:             end for
12:         end if
13:     end for
14: end function
15:
16: function GETBOOLINV($T, l, u$)
17:     $inv\_result[2] = \{0\}$ ;
18:     for $t = 0$; $t <$getSize($T$)$-l+1$; $t = t+1$ do
19:         $local\_res[2] = \{1\}$;
20:         for $s = 0$; $s < l$; $s = s+1$ do
21:             $u\_val$=getValue($T$, $t+s$, $u$);
22:             $local\_res[0]$=$local\_res[0] \wedge (u\_val$==false);
23:             $local\_res[1]$=$local\_res[1] \wedge (u\_val$==true);
24:             if allZero($local\_res$) then //optimization 1
25:                 break;
26:             end if
27:         end for
28:         $inv\_result = inv\_result \vee local\_res$;
29:         if allOne($inv\_result$) then //optimization 2
30:             break;
31:         end if
32:     end for
33:     return $inv\_result$;
34: end function
35:
36: function GETNUMINV($T, l, u, v$)
37:     $inv\_result\ [6] = \{0\}$ ;
38:     for $t = 0$; $t <$getSize($T$)$-l+1$; $t = t+1$ do
39:         $local\_res\ [6] = \{1\}$;
40:         for $s = 0$; $s < l$; $s = s+1$ do
41:             $u\_val$=getValue($T$, $t+s$, $u$);
42:             $v\_val$=getValue($T$, $t+s$, $v$);
43:             for $i = 0$; $i <$getSize($I_{Num}$); $i = i+1$ do
44:                 $local\_res[i]$=$local\_res[i] \wedge$check($inv\_i, u, v$);
45:             end for
46:             if allZero($local\_res$) then //optimization 1
47:                 break;
48:             end if
49:         end for
50:         $inv\_result = inv\_result \vee local\_res$
51:         if allOne($inv\_result$) then //optimization 2
52:             break;
53:         end if
54:     end for
55:     return $inv\_result$;
56: end function
```

the time window for invariant inv_i (lines 22-23 and 43-45).

3. At the end of the time window analysis, *inv_result* is updated according to the value of *local_result* (lines 28 and 50). If *local_result* is 1 then also *inv_result* becomes 1 to store that the algorithm found a time windows where inv_i holds.

The number of checks performed by the algorithm (i.e., lines 22 and 23 for the Boolean variables, and line 44 for the numeric variables), in the worst case, depends on:

- the number of variables' pairs to be checked (i.e., $|V|^2$, for the 2-ary invariants considered in the paper);
- the length of the time-window (i.e., l), and consequently the number of time windows in the execution trace (i.e., $(length(T) - l + 1)$); and
- the total length of the execution trace (e.g., $length(T)$).

Thus, according with the previous considerations, the algorithm scales best in $|T|$ for very small time windows or those close to $|T|$. To reduce the overall execution time, we have introduced two optimizations. The first concerns the checking of invariants within a single time window TW. The iteration on all the simulation instants of TW exits as soon as a counter example for each invariant has been found (lines 24-25 and 46-47). This optimization generally reduces execution time in case of very long time windows that expose very few invariants. The second optimization concerns the checking of invariants by considering all time windows of the same execution trace T. The iteration on all the time windows of T exits as soon as all invariants have been verified on at least one time window (lines 29-30 and 51-52). This optimization reduces execution time in case of very long execution traces with several time windows and high probability of finding time windows where the candidate invariants hold.

4.2 Parallel algorithm

According to the considerations reported at the end of the previous section, the sequential algorithm is efficient when the length of the execution trace T is low and the corresponding time windows are either short or almost as long as T. On the contrary, the worst cases occur for a very long execution trace T with time windows whose length is near to the half of the length of T. To preserve efficiency even in cases where invariant mining becomes unmanageable by using the sequential algorithm, we defined also a parallel version that can be executed by a GPU.

The parallel algorithm works on a *mining matrix* M of $|V|*|V|$ unsigned integers. The set of variables V of a model is partitioned in two subsets V_{Bool} and V_{Num} that contain, respectively, Boolean and numeric variables. The binary representation of the unsigned integer stored in each element $M[i][j]$, with $i \neq j$, corresponds to the *invariant vector* of the pair $(v_i, v_j) \in V_{Num} \times V_{Num}$. Each element on the diagonal $M[k][k]$ corresponds to the invariant vector of a Boolean variable $v_k \in V_{Bool}$. Elements $M[i][i]$ with $v_i \in V_{Num}$ and $M[i][j]$ with either $v_i \in V_{Bool}$ or $v_j \in V_{Bool}$ are not used. Elements $M[j][i]$ below the diagonal are not used too, since they would contain the dual of the invariant vector stored in $M[i][j]$. In summary, $\big(|V_{Bool}| + (|V_{Num}| * (|V_{Num}| - 1))/2\big)$

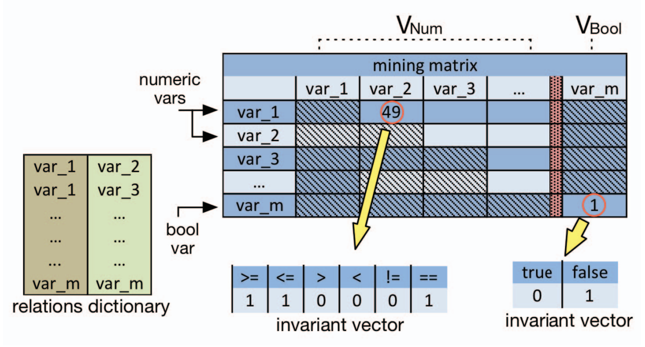


Figure 4: A relation dictionary and the corresponding mining matrix. Shaded elements are not used. The diagonal is used for Boolean variables.

elements of M are active during the execution of the algorithm. A list of the variable pairs corresponding to the active elements of the mining matrix is stored in a *relation dictionary*. Figure 4 shows an example of a relation dictionary and the corresponding mining matrix.

The mining matrix can be generalized to an n-dimensional array to mine logic relations with arity till n. For example, to mine unary, binary and ternary templates, a three-dimensional array (a cube) should be used to represent all the possible combination of three variables. In this case, unary relations will be stored in the element $M[i][i][i]$ (diagonal of the cube), binary relations will use only the faces of the cube, and ternary relations also internal elements. For simplicity and without loss of generality, in the following we consider only unary relations on Boolean variables and binary relations on numeric variables.

The execution flow of the parallel approach can be summarized in three steps:

1. create and copy the mining matrix and the relation dictionary into the global memory of the GPU;
2. run a parallel kernel in the GPU to extract invariant;
3. read the mining matrix from the global memory and print the results.

In order to achieve a better performance we defined two different kernel implementations for step 2: one for mining T-invariant (*check_T-invariants*) according to conditions C3 and C4 defined at the beginning of this section, and one for mining TW-invariants (*check_TW-invariants*) according to conditions C1 and C2.

4.2.1 Mining of T-invariants

The $check\_T - invariant$ kernel searches for invariants that are true in every simulation instant of an execution trace. The kernel takes advantage of the efficient allocation of threads in the GPU. The idea behind the approach is:

- to instantiate in the GPU as many *thread blocks* as the number of entries of the relation dictionary (i.e., each block works on a different entry of the relation dictionary), and
- to instantiate for each block the maximum number of available threads (e.g., 1024 threads in case of the GPU we used for experimental results).

Every thread of a block checks in parallel to the other threads of the same block if each of the considered invariant templates is true for the target entry of the relation dictionary in a precise simulation instant t of the execution trace (i.e., each thread works in parallel on different simulation instants). The approach to verify if an invariant template holds on a pair of variables is exactly the same implemented in functions $getBoolInv$ and $getNumInv$ (see Section 4.1). After checking, each thread updates the corresponding invariant vector into the *mining_matrix* (the elements of the matrix are at the beginning initialized with 1). In particular, the thread that works on pair (v_i, v_j) for a simulation instant t stores the result in element $M[i][j]$ by means of an *AtomicAnd* operation, which is executed sequentially with respect to other *AtomicAnd* performed by different threads that work on the same pair (v_i, v_j) but on different simulation instants. In this way, when all threads complete the kernel execution, the number stored in $M[i][j]$ represents the final invariant vector of (v_i, v_j) over the execution trace. The same considerations apply for elements of kind $M[k][k]$ related to each Boolean variable v_k.

Moreover, to increase the efficiency of the parallel approach, the following optimizations have been implemented:

- The execution trace is partitioned in *slices* which are asynchronously loaded into the GPU global memory. To achieve better performance we used different streams (i.e., *cudaStream*) to asynchronously load and elaborate different slices of the execution trace.
- If the threads of a block falsify all invariant templates for an entry of the relation dictionary in one slice of the execution trace, they do not check the same invariant templates in the subsequent slices of the same execution trace. This does not create divergence on threads because all the threads of a block deal with the same entry of the relation dictionary.

Figure 5 graphically shows how the threads of different blocks can work in parallel, on different entries of the relation dictionary and different time intervals, to speed-up the invariant checking. For example, block(0,0) works on simulation instants belonging to the interval [0, 1023] for the entry (u, v), while block (0,1) works on the same interval but for the entry (u, z), and block(1,0) works on the same entry (u, v) of block (0,1) but on the interval [1024, 2047].

4.2.2 Mining of TW-invariants

The $check\_TW - invariant$ kernel searches for invariants that are true in at least one time window of an execution trace. The idea behind the approach is basically the same as for the $check\_T - invaraint$ kernel, i.e., to assign an entry of the relation dictionary to every block of threads. However, two aspects differentiate $check\_TW - invariant$ from $check\_T - invariant$:

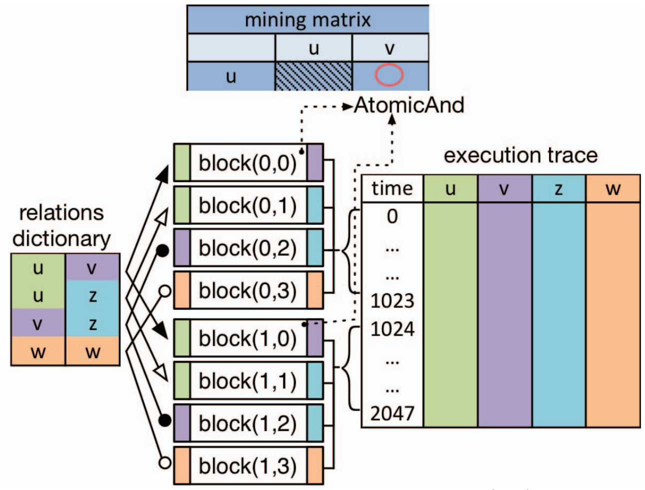


Figure 5: Allocation of thread blocks. Block(i, j) works on dictionary entry j by analysing slice i of the execution trace.

- each thread of the same block checks if invariant templates are true on a *different time window* of the same execution trace (not on a different time instant);
- the thread that works on the entry (v_i, v_j) of the relation dictionary for a given time window stores the result in element $M[i][j]$ of the mining matrix (the elements of the matrix are at the beginning initialized with 0) by means of an *AtomicOr* operation. This guarantees that at the end of the procedure, each element of the invariant vector stored in $M[i][j]$ is set to 1 if the corresponding invariant template has been satisfied by at least one time window.

Furthermore, to increase the efficiency of the parallel approach, the following optimizations have been implemented:

- Since all threads of the same block analyse the same entry of the relation dictionary on overlapping time windows, the currently-analysed slice of the execution trace is copied in the GPU shared memory. This greatly reduces the time required for retrieving the value of analysed variables, since the latency of the shared memory is really lower than the latency of the GPU global memory. For example, Fig. 6 shows how a block of 1024 threads works to check invariant templates for the dictionary entry (u, v). First, values assumed by u and v on a slice of the execution trace (e.g., simulation instants in the interval [0, 2047]) are copied into the shared memory. Then, all threads of the block check the invariant templates on different time windows with the same length. Each time window starts one simulation instant later than the precedent time window. If a time window exceeds the slice, new data are shifted from the execution trace into the shared memory and the verification process is resumed. When all time windows have been analysed, every thread stores its local result into the mining matrix through an *AtomicOr*.
- In case the currently-analysed time window exceeds the slice of the execution trace loaded in the shared memory, as soon as all invariant templates have been falsified, the block of threads stops to check the same invariant templates on the following slices. This does not create divergence on threads because all the threads of a block deal with the same entry of the relation dictionary.

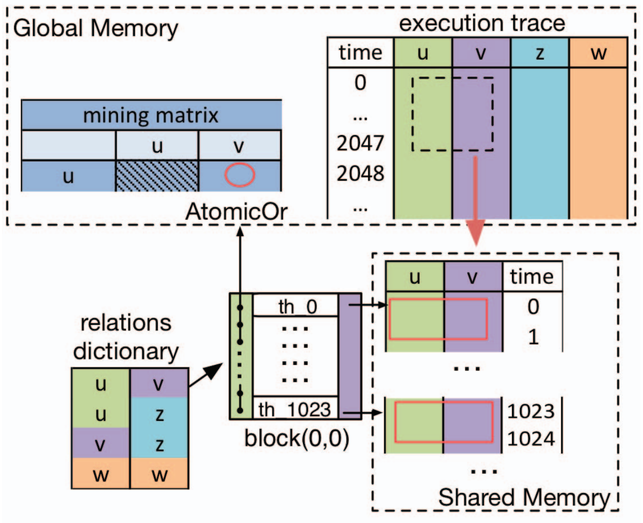


Figure 6: Use of the shared memory to speed up mining of TW-invariants.

5. EXPERIMENTAL RESULTS

The sequential and the parallel approaches have been evaluated in comparison with Daikon version 5.2.0. For a fair comparison, Daikon has been configured such that it searched only for the same invariant templates implemented in our algorithms. This restriction does not affect the fairness of the comparison. In fact, the inclusion of a larger set of invariant templates would have the same effect on our algorithms as well as on Daikon, i.e., the verification time would increase proportionally with the number of invariant templates to be checked. The extension to the full set of Daikon's template is an ongoing activity.

For all experiments, our approaches and Daikon extracted the same set of invariants. Thus, from the accuracy point of view they are equivalent, while they differ from the execution time point of view. Performances have been evaluated on execution traces with different characteristics by running experiments on an AMD Phenom II X6 1055T (3GHz) host processor equipped with 8.0GB of RAM, running Linux OS, and connected to an NVIDIA GEFORCE GTX 780 with CUDA Toolkit 5.0. Results are reported for mining both T-invariants, covering conditions C3 and C4 of Section 4, as well as TW-invariants, covering conditions C1 and C2.

5.1 Execution time for mining T-invariants

The type of SUV (i.e., HW design or SW program), and the complexity of the SUV (in terms, for example, of memory elements, lines of code, cyclomatic complexity) are not particularly relevant to measure the performance of approaches for dynamic invariant mining. The analysis of a long execution trace exposing several invariants among variables, even if corresponding to a functionally simple SUV, may require

Trace length	Numeric variables	Boolean variables	Invariants number	Daikon time (s.)	Sequential time (s.)	Parallel time (s.)
1000000	15	15	0	27.3	2.8	3.2
3000000	15	15	0	74.4	8.5	9.1
5000000	15	15	0	118.3	13.9	14.9
1000000	10	10	0	21.6	2.0	2.5
1000000	30	30	0	47.6	5.6	6.4
1000000	50	50	0	73.9	9.1	9.3
1000000	15	15	120	20.3	6.5	3.3
3000000	15	15	120	51.6	19.7	8.8
5000000	15	15	120	82.3	32.8	14.8
1000000	10	10	55	15.4	2.9	2.3
1000000	30	30	465	35.2	25.6	6.1
1000000	50	50	1275	58.5	80.7	10.5

Table 1: Execution time (in seconds) to mine T-invariants from execution traces *with* and *without* invariants at varying of the trace length and the variable number.

Time window length	Numeric variables	Boolean variables	Invariants number	Daikon time (s.)	Sequential time (s.)	Parallel time (s.)
100000	50	50	0	$\approx 141\times10^5$	2378.9	39.5
500000	50	50	0	$\approx 209\times10^5$	1324.8	26.1
900000	50	50	0	$\approx 69\times10^5$	272.1	12.9
5	50	50	1275	$\approx 42\times10^5$	128.4	10.9
25	50	50	1275	$\approx 43\times10^5$	312.2	11.1
100000	50	50	1275	$\approx 326\times10^5$	$\approx 147\times10^4$	2887.8
500000	50	50	1275	$\approx 778\times10^5$	$\approx 832\times10^4$	8075.7
900000	50	50	1275	$\approx 273\times10^5$	$\approx 333\times10^4$	2949.6

Table 2: Execution time (in seconds) to mine TW-invariants from an one-million-long execution trace (violet rows refer to the best cases where time windows are short; the red row refers to the worst case where the length of the time windows is half of the trace).

much more time than a shorter execution trace of a very complex SUV. Indeed, execution time of invariant mining depends on the number and length of the execution traces to be analysed, the number of considered variables, and the number of invariants actually present in the traces (due to the effect of the two optimizations described at the end of Section 4.1. Thus, experimental results have been conducted on randomly generated execution traces with different values for such parameters by considering boolean and numeric (integer and real) data-type variables.

Table 1 reports the time[2] spent by the three approaches (Daikon, sequential algorithm and parallel algorithm) to analyse execution traces from which no invariant (above the central double line) and several invariants (below the central double line) can be mined (see Column *Invs*). For traces without invariants (but we observed the same behaviour in case of very few invariants), our sequential and parallel approaches present similar execution times, which are one order of magnitude lower than Daikon's time. The speed-up achieved by the parallel algorithm thanks to the use of the GPU, is compensated in the sequential algorithm by optimization 1 (lines 26 and 46 of Algorithm 1), which allows the sequential algorithm to discard the entire execution trace as soon as all invariant templates have been falsified.The parallel algorithm, on the other hand, partially benefits from this optimization, since it must elaborate at least an entire slice of the execution trace. When the number of invariants that can be mined in the trace increases, the effect of optimization 1 decreases, thus the parallel algorithm becomes the most efficient solution thanks to its capability of analysing in parallel several instants of the execution trace.

[2]Reported execution times include also the time required to read the execution trace and print the list of mined invariants. This time is not negligible and it is practically the same for the three approaches. Its removal would further amplify the difference among the scalability of the approaches.

5.2 Execution time for mining TW-invariants

The second experiment shows the performance of the two proposed approaches compared to Daikon for mining TW-invariants on at least one time window of an execution trace. This analysis is more time consuming than mining T-invariants on the whole execution trace, since a huge number of partially overlapping time windows must be iteratively analysed. This is necessary, for example, for temporal assertion miners, where invariants extracted from different time windows are composed of means of temporal operators to create temporal assertions that hold on the whole execution trace [6].

Table 2 shows the results at varying time window lengths, by considering an execution trace with one million instants. When the length of time windows is low (violet rows), the sequential and the parallel algorithms require, respectively, few minutes and few seconds to complete the analysis, while Daikon is up to five orders of magnitude slower. For long time windows (hundreds of thousand of simulation instants), the parallel approach is two orders of magnitude faster than the sequential algorithm and three than Daikon. The worst case, as expected, occurs when the length of the time windows is half of the execution trace and the number of invariants is high (red row). It actually takes a couple of hours with the parallel algorithm, while it would take about 3 months with the sequential algorithms and 6 months with Daikon. Indeed, execution times reported for Daikon, and part of those of the sequential algorithm (highlighted by symbol $\approx$) have been estimated according to values achieved on shorter execution traces, because it would be unmanageable to run real experiments. The parallel algorithm, on the other hand, scales very efficiently also in these cases.

6. CONCLUDING REMARKS

An efficient approach for dynamically mining invariants from execution traces has been proposed. The use of array-based data structures for encoding mining information allowed us to implement an efficient mining procedure that can be further optimized to exploit parallel-programming paradigms on a GPU. The approach outperforms a state-of-the-art tool like Daikon by using both the sequential and the parallel versions. In particular, the parallel algorithm allows to efficiently mine invariants by considering tens of variables on millions of time windows composed of millions of simulation instants. Future works will be devoted to further improve the parallel approach by adopting optimized parallel patterns (e.g., reduction procedures) [12], and to extend the set of supported invariant templates by including generic n-ary arithmetic logic relations.

7. REFERENCES

[1] http://docs.nvidia.com/cuda.

[2] http://plse.cs.washington.edu/daikon/pubs.

[3] http://www.khronos.org/opencl.

[4] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *J. of Object Technology*, 5(5):31–58, 2006.

[5] C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proc. of ACM/IEEE ICSE*, pages 422–431, 2005.

[6] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. of ACM/IEEE DATE*, pages 1–6, 2015.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[8] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2-4):97–108, 2001.

[9] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proc. of ACM/IEEE DAC*, pages 775–778, 2005.

[10] R. Hastings and B. Joyce. Joyce. purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter USENIX Conference*, 1991.

[11] R. D. Marat Boshernitsan and A. Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. of ISSTA*, pages 169–180, 2006.

[12] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[13] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proc. of ACM FSE*, pages 11–20, 2002.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *ACM Trans. on Computer Systems*, pages 391–411, 1997.

[15] M. S. L. Sudheendra Hangal. Tracking down software bugs using automatic anomaly detection. In *Proc. of ACM/IEEE ICSE*, pages 291–301, 2002.

[16] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, volume 4260 of *LNCS*, pages 717–736. Springer, 2006.