# Mangrove: an Inference-based Dynamic Invariant Mining for GPU Architectures

Nicola Bombieri* Federico Busato*, Alessandro Danese*, Luca Piccolboni†, and Graziano Pravadelli*‡
*Department of Computer Science, University of Verona, Italy. Email: name.surname@univr.it
†Department of Computer Science, Columbia University, New York, USA. Email:piccolboni@cs.columbia.edu
‡EDALab s.r.l., Italy. Email: name.surname@edalab.it

**Abstract**—Likely invariants model properties that hold in operating conditions of a computing system. Dynamic mining of invariants aims at extracting logic formulas representing such properties from the system execution traces, and it is widely used for verification of intellectual property (IP) blocks. Although the extracted formulas represent likely invariants that hold in the considered traces, there is no guarantee that they are true in general for the system under verification. As a consequence, to increase the probability that the mined invariants are true in general, dynamic mining has to be performed to large sets of representative execution traces. This makes the execution-based mining process of actual IP blocks very time-consuming due to the trace lengths and to the large sets of monitored signals. This article presents *Mangrove*, an efficient implementation of a dynamic invariant mining algorithm for GPU architectures. *Mangrove* exploits inference rules, which are applied at run time to filter invariants from the execution traces and, thus, to sensibly reduce the problem complexity. Mangrove allows users to define invariant templates and, from these templates, it automatically generates kernels for parallel and efficient mining on GPU architectures. The article presents the tool, the analysis of its performance, and its comparison with the best sequential and parallel implementations at the state of the art.

**Index Terms**—Invarinant Mining, GPUs, Inference.

## 1 Introduction

Invariants model stable conditions and behaviours of a computing system. They are represented through logic formulas over variables on observation points (e.g., interface or internal signals of an IP model), and hold along the system execution. Such a formal representation of the system behaviour is widely used in the verification phase of both SW and HW IP models at different abstraction levels. Some examples are the analysis of dynamic memory consumption [1], identification of memory access violations [2], static checking [3], [4], detection of race conditions [5], mining of temporal assertions [6], [7], control-flow error detection [8], test generation [9], and bug catching in general [10].

The extrapolation of invariants (called *invariant mining*) of an IP model is performed statically or dynamically. Static invariant mining exhaustively provides solutions by exploring the state space of the IP model [11]. Nevertheless, it requires the IP model source code and, suffering from the state explosion problem, it can be applied to small systems [12].

Dynamic invariant mining is the solution when the source code is not available or when the IP model size is realistically large [10], [13], [14], [15]. Dynamic mining starts from a set of *candidate invariants* and checks them on the model execution traces. An execution trace describes the values assumed by observed variables along every instant of the model simulation (see Figure 1 for an example with five variables over ten simulation instants). Dynamic invariant miners, generally, work by analysing a set of execution traces of the DUV searching for counterexamples of the logic formulas that represent the desired invariant candidates. A formula holding till the end of the simulated execution traces is collected, while in case

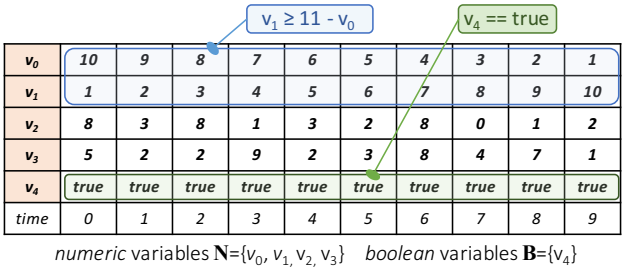


Fig. 1. Example of execution trace and two likely invariants.

it fails in at least one simulation instant, it is discarded. At the end, survived candidates represent the final set of *likely invariants* (e.g., $v_1 \geq 11 - v_0$ and $v_4 == true$, in the example), i.e., formulas that are true throughout the analysed execution traces, and then, that are statistically true on the design under verification (DUV). However, it can happen that a collected likely invariant does not always hold on the DUV. If the set of analysed traces is incomplete, the miner can ignore the presence of a trace where the mined invariant fails. Thus, the quality of the mined invariant depends on the quality of the analysed execution traces. Higher is the number of behaviour exported by the analysed traces, higher is the probability that the likely invariant always hold on the DUV.

Consequently, to extrapolate robust likely invariants of an IP model and thus maximizing the probability that they are always true for the DUV, even for not analysed execution traces, dynamic mining has to be performed on large and representative sets of execution traces. In real case studies, this means running the process on thousands of execution

traces, including millions of clock cycles, and over hundreds of variables, which can lead to prohibitive execution times. However, existing dynamic miners are effective in the analyses of short sequences of events, restricted to specific parts of the design, rather than a large set of very long execution traces, since they suffer from scalability issues when applied to large data. Unfortunately, in many scenarios, especially for the HW domain from the RTL level and below, many verification procedures require the analysis of long and extensive simulation runs, searching for formulas that always hold.

Motivated by the previous considerations, this article presents *Mangrove*, a tool for flexible and dynamic mining of likely invariants, which outperforms, in terms of execution times and scalability, existing miners by exploiting parallel programming and effective inference rules to search for invariants in execution traces of large designs lasting millions of simulation cycles. In particular, it relies on three key points: (i) it allows users to define customized templates for generating candidate invariants thus making the approach flexible and portable to different IP characteristics; (ii) it implements *inference rules* to get rid of redundant invariants thus sensibly reducing the mining effort; (iii) it has been implemented for parallel and efficient execution on GPU architectures.

To show the effectiveness and efficiency of *Mangrove*, the article presents the experimental results obtained by applying the proposed solution to a set of IP models, and a comparison with the most representative sequential and parallel tools for dynamic invariant mining at the state of the art [13], [16]. Beside being more flexible and portable in supporting invariants, *Mangrove* allows reducing the mining time up to one order of magnitude with respect to the best parallel solution for GPUs [16] and up to two orders of magnitude with respect to the best sequential solution [13] at the state of the art. Such a significant reduction in the execution time allows Mangrove to solve scalability issues of previous approaches and manage invariant mining for large designs in the context of extensive simulation runs.

*Mangrove*, which is now available on github (https://github.com/mangrove-univr/Mangrove), is the complete and extended implementation of the work preliminary presented in [17]. With respect to [17], this article presents the following novel contributions:

- A complete framework that, starting from invariant templates defined by the user, automatically generates optimized kernels for execution on GPU devices. The new approach relies on advanced techniques to better exploit the GPU massive parallelism, such as efficient parallel data reading from memory, control-flow organization and coordination of GPU threads, and GPU kernel autotuning.
- A technique to deal with the large sizes of execution traces, which implements data parsing from the GPU device.
- A technique that implements a sophisticated inference-based invariant miner to exploit input data characteristics, statistics, and inference rules with the aim of efficiently dealing with very large execution traces while sensibly reducing the mining effort.

The article is organized as follows. Section 2 gives some background and definitions on invariant mining. Section 3 presents and accurate analysis of the related work. Section 4 presents an overview of *Mangrove*. Section 5 presents the details about the GPU kernel generation, while Section 6 describes an optimization for reading massive data on the GPU device. Section 7 presents the inference rules. Section 8 presents the experimental results, while Section 9 is devoted to the concluding remarks.

## 2 Background

In the context of dynamic mining of *likely invariants* (*invariants* in the rest of the paper) we consider the following definitions:

**Definition 2.1.** Given a model $\mathcal{M}$ working on a set of variables $\mathcal{V} = \{v_0, v_1, .., v_n\}$ and a finite sequence of simulation instants $\mathcal{T} = \langle t_0, ..., t_m \rangle$, an **execution trace** of $\mathcal{M}$ is a finite sequence of pairs $E = \langle (v_0, t_0), (v_0, t_1), .., (v_n, t_m) \rangle$ generated by simulating $\mathcal{M}$, where $(v_i, t_j)$ is the evaluation of variable $v_i$ at the simulation instant $t_j$.

More informally, an execution trace describes the values assumed by each variable in $\mathcal{V}$ along every instants of the simulation time $\mathcal{T}$. Figure 1 shows an example of execution trace $E$, in which $\mathcal{V} = \langle v_0, v_1, v_2, v_3, v_4 \rangle$ and $\mathcal{T} = \langle t_0, ..., t_9 \rangle$.

In this work, we assume that each variable in $\mathcal{V}$ is either a *boolean*, or a *numeric* variable[1]. We refer to $\mathcal{B}$ as the set of *boolean* variables, and $\mathcal{N}$ as the set of *numeric* variables.

**Definition 2.2.** Given an execution trace $E$ and the corresponding set of variables $\mathcal{V}$, an **invariant** is a formula over $\mathcal{V}$ that always holds along the whole execution trace.

In Fig. 1, the formulas $v_1 \geq 11 - v_0$ and $v_4 == true$ are examples of invariants for the considered execution trace $E$.

**Definition 2.3.** Given the set of variables $\mathcal{V}$ of an execution trace $E$ and the set of all possible permutations of $k$ variables $\mathcal{P}_k$ (e.g., $\mathcal{P}_2 = \{(v_0, v_1), (v_0, v_2), .., (v_1, v_0), (v_1, v_2), .., (v_4, v_3)\}$, a **dictionary** is a set of entries $\mathcal{D}_k = \{d_1^k, ..., d_n^k\}$, where $\mathcal{D}_k \subseteq \mathcal{P}_k$.

Figure 2(a) shows some examples of dictionary ($\mathcal{D}_1, \mathcal{D}_2$, and $\mathcal{D}_3$) considering the *numeric* variables of Figure 1.

**Definition 2.4.** Given a data type (i.e., boolean or numeric), a **template variable** is a place holder in a formula that can be substituted with any variable of the same type.

Figure 2(b) shows some examples, in which $bVar$ is a template variable for the *boolean* type and $nVar_0$, $nVar_1$, and $nVar_2$ are template variables for the *numeric* type.

**Definition 2.5.** An **invariant template** (*template* in the follows) is a formula composed of arithmetic/logic operators, constants, and template variables having the same type.

Figure 2(b) shows three examples of invariant templates, which involve one *boolean* variable template ($bVar$), three *numeric* variable templates ($nVar_0$, $nVar_1$, and $nVar_2$), the numeric constant 11, the boolean constant `true`, and the arithmetic/logic operators "`-`", "`+`", and "`==`". In the rest of the paper, we refer to $k$-arity invariant template as a template involving $k$ different template variables. The

1. For the sake of clarity and without loss of generality, we consider a *numeric* variable as a floating-point data type

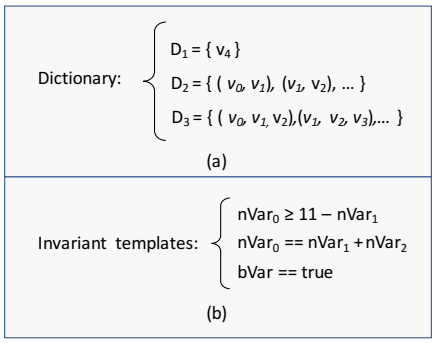


Fig. 2. Examples of dictionary (a) and invariant templates (b).

| Mining Tool | Application Domain | Data Types | | Main Features | | |
|---|---|---|---|---|---|---|
| | | Boolean | Numeric | Code Indep. | Custom Templates | Parallel |
| **Daikon** [13] | SW | ✓ | ✓ | ✓ | limited | ✗ |
| **Agitator** [15] | SW | ✓ | ✓ | ✓ | limited | ✗ |
| **DIDUCE** [10] | SW | ✓ | ✓ | ✗ | limited | ✗ |
| **IODINE** [14] | HW | ✓ | ✗ | ✗ | limited | ✗ |
| **Turbo** [16] | SW/HW | ✓ | ✓ | ✓ | ✗ | ✓ |
| **Mangrove** | SW/HW | ✓ | ✓ | ✓ | ✓ | ✓ |

TABLE 1
Classification of the state-of-the-art approaches (✓ fully supported - ✗ not supported)

invariant templates in Figure 2(b) are examples of 2-arity, 3-arity, and 1-arity templates, respectively.

**Definition 2.6.** According to the above definitions, we define the **instantiation** of a $k$-arity invariant template with an entry $d \in \mathcal{D}_k$ as a procedure that replaces the $i$-th template variable in the template with an actual variable of $d$, for each $i$ $(0 < i < k)$.

Considering for example the invariant template $nVar_0 = 11 - nVar_1$ and the entry $d_0^2 : (v_0, v_1) \in \mathcal{D}_2$ in Figure 2(b) and Figure 2(a), respectively, the instantiation of the template with $d_0^2$ results in the invariant $v_0 = 11 - v_1$.

## 3 Related work

Several approaches for mining invariants have been proposed in the last years. The most efficient and widepread is *Daikon* [13]. Such a dynamic tool analyses execution traces of a software application through an inference engine that *incrementally* detects invariants. Through a configuration file, the user specifies which invariant templates, from a pool of predefined templates, have to be enabled during the mining phase. *Daikon* collects invariants from execution traces by (i) instrumenting the software application, through a language-specific instrumenter, to trace the variables of interest, (ii) executing the instrumented code with a set of test cases, and (iii) checking the invariants over the values computed by the Software under evaluation. To extract invariants, *Daikon* applies an incremental approach, which relies on an invariant-specific optimization that reduces the number of expressions to be verified. *Daikon* computes also the *invariant probability*, that is, the probability for an invariant to appear in a random trace. An invariant is reported if the computed probability is lower than a user-defined threshold.

Alternatives to *Daikon* have been reviewed in [18]. The most efficient include *Agitator* [15], which is a commercial tool that includes a dynamic invariant engine and other mechanisms to test software. The types of extracted invariants are similar to the *Daikon* ones, yet the approach is based on some heuristics. The *miner* filters the hypothetical relations and, through an *agitation phase*, it discards the invariants that do not hold always. *DIDUCE* [10] is a tool that aims at finding complex errors in `Java` programs. Like *Daikon*, it tries to extract invariants dynamically from execution traces, but unlike *Daikon*, it continually checks the program's behavior and reports all detected violations. When a dynamic invariant violation is detected, the invariant is relaxed to represent a new behavior. Another alternative is represented by *IODINE* [14], which allows automatically extracting hardware invariants from design simulations. *IODINE* performs a multi-pass analysis on the execution traces by running a set of analyzers, which are invoked at user-specified trigger events. Each analyser saves the inferred invariants in a database, assigning them a confidence level by using an invariant-specific policy similarly to *Daikon*.

The first attempt that tries to exploit GPU architectures for invariant mining is *Turbo* [16], which extracts the invariants by offloading the verification of each invariant template to a different GPU thread. It outperforms *Daikon* from the performance point of view, but it supports a very restricted set of invariant templates.

Table 1 summarizes the main characteristics of all these existing mining tools, and it compares them to the solution proposed in this work (*Mangrove*). The table reports the application domains, the supported data types, and the main important features of the tools. The features include the independence from the source code of the software program or hardware design (Code Indep.), the possibility to extract custom invariant templates (Custom Templates), and the possibility to exploit the intrinsic parallelism of the mining algorithm (Parallelism). As evidenced by the table, while most of the approaches support both boolean and numeric types, none of them has a full support for custom templates. Most approaches allow for the extraction of custom invariants, although they require a manual implementation of the invariant and they allows using a limited number of variables. Only *Turbo* exploits parallelism, but it only supports a subset of invariant templates. In contrast, *Mangrove* implements all these important features, as explained in the following sections.

## 4 Mangrove overview

*Mangrove* implements the invariant mining algorithm shown in Algorithm 1. Starting from an execution trace $E$ and a set of invariant templates $\mathcal{I}$, it first generates a dictionary $(\mathcal{D}_k)$ and, then, it combines each entry $d \in \mathcal{D}_k$ with each invariant template $it \in \mathcal{I}$ (lines 4-5). In particular, for each $d$ and for each $it$, it instantiates a dictionary entry with $it$ by replacing the variable templates in $it$ with a variable of $E$ belonging to $d$. This results in a set of *candidate* invariants (*inv* in line 7). It evaluates every candidate invariants along the simulation instants $t \in \mathcal{T}$ and it discards any candidate as soon as such a candidate does not hold. The result, after $|\mathcal{T}|$ simulation instants, is the set of invariants $\mathcal{L}$ for E.

The base algorithm has a worst-case time complexity equal to $\mathcal{O}\big((|\mathcal{V}|!/|\mathcal{V}-k|!) \cdot |\mathcal{I}| \cdot |\mathcal{T}|\big)$, where $|\mathcal{V}|$ is the number of variables in $E$, $k$ is the number of variables in each entry of $\mathcal{D}$,

**Algorithm 1** Invariant mining algorithm

```
   Input: execution trace E,
          set of invariant templates I
   Output: set of invariants L
1: L = ∅
2: for each k do
3:    Generate dictionary D_k
4:    for each d ∈ D_k do
5:       for each it ∈ I do
6:          flag = true
7:          inv = instantiation of it with d
8:          for each instant t ∈ E do
9:             flag = flag AND eval(inv, E, t)
10:         end for
11:         if flag == true then
12:            L = L ∪ inv
13:      end for
14:   end for
15: end for
```

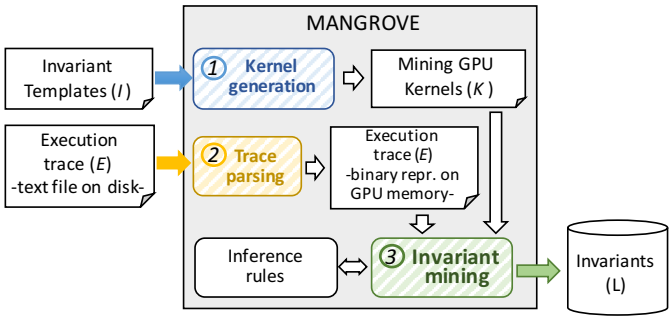


Fig. 3. Overview of *Mangrove*.

$|\mathcal{I}|$ is the number of invariant templates in $\mathcal{I}$, and $|\mathcal{T}|$ is the number of simulation instants of $E$. Given the complexity of the problem, *Mangrove* implements Algorithm 1 for parallel execution on GPU architectures. Such a parallel invariant mining consists of three main phases (see Fig. 3):

1) **Kernel generation:** For each user-defined invariant template, *Mangrove* automatically generates a *checker* that evaluates the template satisfiability. The checkers are defined as code lines and integrated in a main GPU kernel. *Mangrove* generates one kernel per dictionary, as explained in Section 5.
2) **Trace parsing:** Since an execution trace in real applications may reach important sizes, the input parsing phase may strongly influence the total mining time. *Mangrove* implements an optimized trace parsing to read the trace file from the disk and to convert it in a binary representation on the GPU memory as explained in Section 6.
3) **Invariant mining:** In the third and last step, *Mangrove* executes the generated kernels on the GPU. The tool exploits *inference rules*, which are applied at run time to filter the invariants from the execution trace and, thus, to improve the tool efficiency as explained in Section 7.

Given an execution trace and a set of invariant templates provided as inputs, Mangrove performs the previous phase in a fully automatic way. The user is only required to provide the tool with a set of invariant templates. No other manual steps are required.

## 5 Kernel generation

Given a set of invariant templates $\mathcal{I}$, the kernel generator automatically synthesizes a *Mining GPU kernel* $\mathcal{K}$, which implements a checker for each invariant template $it \in \mathcal{I}$.

### 5.1 Invariant template grammar

We define the *invariant template grammar* to formally specify which kinds of invariants *Mangrove* can mine from an execution trace $E$, as follows:

```
1: templates  := template | templates template
2: template   := bInvariant | nInvariant
3:
4: bInvariant := bVar == true | bVar == false
5:               | bVar == bFunction
6: bFunction  := bTerm bOperator bFunction | bTerm
7: bOperator  := ∧ | ∨ | ⊕
8: bTerm      := bVar | ¬bVar
9:
10: nInvariant := nVar nRelation rExpr
11: nRelation  := ≤ | < | ≠ | ==
12: rExpr      := nTerm | uFunc(nVar)
13:               | nFunc(nVar, rExpr)
14: nTerm      := nVar | nConst
15:
16: uFunc      := <unary user-defined function>
17: nFunc      := <binary user-defined function>
```

A boolean invariant template (`bInvariant`, line 4) always starts with a `bVar` and the equality operator ($==$). Then, `bInvariant` includes either a *boolean* constant, or a `bFunction` (line 6). A `bFunction` always starts with a `bTerm` (line 8), which can be either a boolean variable or the complement of a boolean variable. `bFunction` ends with a `bOperator` followed by a `bFunction`. A `bOperator` (line 7) can be one of the following operators: $\wedge$, $\vee$, $\oplus$.

Similarly, a numeric invariant template (`nInvariant`, line 10), always starts with a `nVar`. `nInvariant` includes a `nRelation` (line 11), which can be one of the following operators: $<$, $\leq$, $>$, $\geq$, $\neq$, $==$. Finally, `nInvariant` ends with a `rExpr` (line 12), which can be either a terminal symbol `nTerm`, or a user-defined numeric constant `nConst`.

*Mangrove* allows introducing a unary function (line 16), and a binary function (line 17). A unary function (`uFunc`) can be any user-defined function taking one input argument, and returning a numeric value as a result. A binary function (`nFunc`) can be any user-defined function taking two numeric values as input and returning a numeric value as a result.

This invariant template grammar allows Mangrove's users to define a richer set of invariants with respect to state-of-the art mining tools. For example, Turbo, the only existing parallel invariant miner, supports a limited set of templates, which is not directly modifiable by the user. In particular, it supports only the following set of templates: $\{u$ op $v$, $u = F$, $u = T\}$, where *op* can be any among $\{=, \neq, <, >, \leq, \geq\}$. Alternatively, Daikon is more flexible compared to Turbo, but the extension of its pre-defined set of template requires users to define new templates in Java by means of custom Java classes. Then, coding is necessary as opposed to Mangrove, where the users define templates through the simple grammar previously described. For example, this allows Mangrove to easily extract relationships, like, $x = y * z + c$, where $x$, $y$ and $z$ are integer variables and $c$ is a constant, by specifying the rule: $nvar == +(*(nVar, nVar), nConst)$. The same is not natively possible with Daikon.

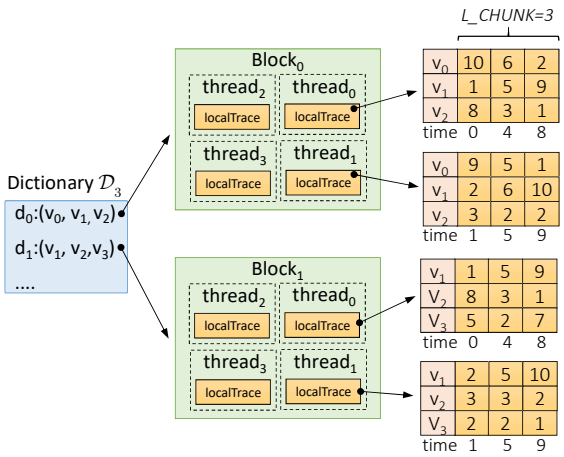


Fig. 4. Workload and execution trace partitioning among blocks and threads of GPU.

### 5.2 Structure of the mining kernels

Algorithm 2 shows the pseudo-code of a standard GPU mining kernel $\mathcal{K}$. Considering the structure of the problem, in which the number of execution instants $|\mathcal{T}|$ is much greater than the number of variables $|\mathcal{V}|$, *Mangrove* implements a workload partitioning with granularity at dictionary entry-level, namely one thread block per entry of $\mathcal{D}_k$ (line 1). This allows all blocks to execute independently, and the threads to independently evaluate the invariant checkers associated to their block over different time instants. The full utilization of the GPU device is guarantee as the number of dictionary entries running in parallel is much greater than the number of thread blocks that can run concurrently on the GPU.

Figure 4 depicts an example of workload partitioning and, in particular, how entries $d_0$ and $d_1$ of dictionary $\mathcal{D}_3$ have been mapped to the thread blocks $Block_0$ and $Block_1$, respectively. For the sake of clarity, each illustrated block consists of four threads, which are individually indexed with the identifier $thread_0$, $thread_1$, $thread_2$, and $thread_3$.

For each entry $d_i$, the shared memory variable `blockResult` (line 2 of Algorithm 2) and the local variable `threadResult` (line 3) are initialized to $\mathcal{I}$. According to Algorithm 1 (line 6), it is initially assumed that each single candidate invariant holds in the whole execution trace $E$. Afterwards, each thread loads a chunk of the execution trace $E$ into its local variable `LocalTrace` through the function LoadTraceChunk. Given in input the execution trace $E$, a simulation instant `t` $\in \mathcal{T}$, and an entry $d_i$ of $\mathcal{D}_k$, function LoadTraceChunk copies a `L_SIZE` fixed-length sequence of values of each variable in $d_i$ from the execution trace $E$ into `LocalTrace`. *Mangrove* exploits coalesced memory accesses during this loading procedure, namely block threads read contiguous values of a same variable from $E$. After this loading phase, each thread evaluates which invariants in `threadResult` hold in `localTrace` through the function $\texttt{templatesEval}_{\langle\mathcal{I}\rangle}$ [2] (line 7). For each invariant template `it` $\in \mathcal{I}$ (line 1), *Mangrove* checks whether `it` $\in$ `threadResult` (line 18). If the check succeeds, the tool evaluates `it` in `LocalTrace` through the function $\texttt{templateEval}_{\langle it\rangle}$ (line 19). In particular, for each simulation instant `t` $\in$ `LocalTrace` (line 28), *Mangrove* evaluates `it` through the function $\texttt{eval}_{\langle it\rangle}$ (line 29), which embeds the invariant checker for the corresponding user-defined invariant templates `it`.

For the sake of clarity, we describe, through the step-by-step example reported in Figure 5, how *Mangrove* recursively generates an invariant checker from an invariant template. Considering, as an example, the invariant template:

`it: nVar0 == nVar1 + nVar2`

the tool applies the invariant template grammar introduced in Section 5.1 to identify each term. `it` is an instance of an `nInvariant` (line 10 of the grammar), since $\texttt{nVar}_0$ is an instance of `nVar`, `==` is an instance of `nRelation`, and $\texttt{nVar}_1$ `+` $\texttt{nVar}_2$ is an instance of `rExpr`. Recursively,

2. *Mangrove* deeply exploits C++11 *variadic* templates and template meta-programming to fully generate and automatically embed invariant checkers at compile-time. The subscripts $\langle\mathcal{I}\rangle$ and $\langle it\rangle$ represent template functions generated by the compiler.

**Algorithm 2** Pseudocode of the mining kernel $\mathcal{K}$

**GPU_Mining_Kernel**( $E$, $\mathcal{D}_k$, $\mathcal{I}$)

**Input:** $E$ execution trace
$\mathcal{D}_k$ dictionary with k-length entries
$\mathcal{I}$ invariant templates
**Output:** $\mathcal{L}$ verified invariant

`BLOCK_SIZE` : thread block size
`L_SIZE` : thread space for storing trace chunk
`BLOCK_CHUNK` : `BLOCK_SIZE` * `L_SIZE`
threadId : thread id
blockId : thread block id
gridDim : number of thread blocks

```
1:  for ( i = blockId; i < |D_k|; i += gridDim) do
2:      blockResult = I                          // shared memory variable
3:      threadResult = I                         // Alg. 1, line 4
4:      Barrier
5:      for ( t = 0; t < |E|; t += BLOCK_CHUNK ) do   // Alg. 1, line 5
6:          LocalTrace[k][L_SIZE] ← LoadTraceChunk(E, t, d_i)
7:          templatesEval_⟨I⟩(LocalTrace, threadResult)
8:          Barrier
9:          threadResult = blockResult
10:         if threadResult = ∅ then     // if all invariants are falsified
11:             break                    // go to the next dictionary entry
12:         Barrier
13:     end for
14:     if threadId = 0 then
15:         L[ blockId ] = threadResult
16: end for
```

**templatesEval**$_{\langle\mathcal{I}\rangle}$( LocalTrace[$k$][L_SIZE], threadResult)

```
17: for each it ∈ I do
18:     if it ∈ threadResult then
19:         EvalResults[L_SIZE] = templateEval_⟨it⟩(LocalTrace)
20:         R = ThreadReduce( EvalResults )
21:         if warp_any(R) == false then
22:             threadResult = threadResult \ it
23:             Update shared value blockResult with threadResult
24:         end if
25:     end if
26: end for
```

**templateEval**$_{\langle it\rangle}$(LocalTrace[$k$][L_SIZE])

```
27: EvalResults[L_SIZE]
28: for each instant t ∈ [0, L_SIZE ] do
29:     EvalResults[t] = eval_it(LocalTrace, t)
30: end for
31: return EvalResults
```

**eval**$_{\langle it\rangle}$( LocalTrace[$k$][L_SIZE], $t$ )

```
32: eval_⟨it−left⟩( LocalTrace[][t], eval_⟨it−right⟩(LocalTrace) )
```

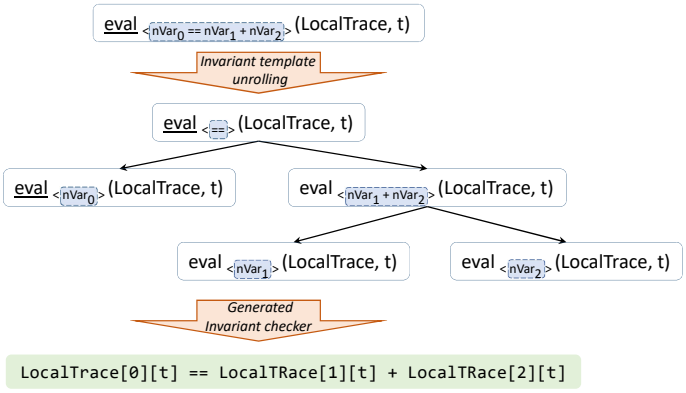


Fig. 5. Generation procedure of an invariant checker given a user-defined invariant template.

we note that $\texttt{nVar}_0$ is a leaf of `it`. In particular, $\texttt{nVar}_0$ represents an access to the values at the row 0 of the `LocalTrace` (i.e. `LocalTrace[0][t]`). The template $\texttt{nVar}_1$ + $\texttt{nVar}_2$ is a `nFunc` implementing the arithmetic operation addition between $\texttt{nVar}_1$ and $\texttt{nVar}_2$. $\texttt{nVar}_1$ and $\texttt{nVar}_2$ are leaves of `it` as well, and they represent the access to the corresponding values in `LocalTrace` (i.e. `LocalTrace[1][t]` and `LocalTrace[2][t]`). In the last step (Fig. 5, bottom-side), the generated expression represents the implementation of the invariant checker for the given invariant template.

After the full evaluation of an invariant template, the predicates for each time instant in the local trace are merged with a sequential thread reduction (line 20). As soon as a thread invalidates a specific invariant, the thread communicates such result to all block threads in two hierarchical steps. First at warp-level with a voting instructions (lines 21, 22), and then at block-level by using a shared memory variable (line 23, 9).

The computation of a given dictionary entry ends when either all template invariants are invalidated (line 10), or a set of invariants have been verified over the whole trace. In the second case, the result is added to the final set of invariants $\mathcal{L}$ (line 15).

It is important to note that the application characterization (memory-bound vs. compute-bound) depends on the user-defined invariant templates. For instance, a binary invariant template involving two variables (e.g., u == v) is memory bound (8 bytes loaded and one operation). More complex templates (e.g., a == ((b * 3 ) + 2) * c) lead to a compute-bound characterization. More in general, the application characterization depends on the number of variables and the operations on them involved in each invariant.

## 5.3 Optimizing the mining kernel for GPUs

The parallel mining process implemented in *Mangrove* aims at reaching high performance and efficiency through three main optimizations:

- Efficient *trace data reading* by parallel threads in the GPU main memory by means of double memory padding, vectorized memory access, and cache interleaving data access (see Section 5.3.1);
- *Control-flow organization and thread coordination* to minimize synchronization overhead and instruction dependencies (see Section 5.3.2);
- *GPU kernel auto-tuning* to identify the best trade-off among device occupancy, synchronization overhead, and thread-level parallelism by considering the characteristics of the target GPU device (Section 5.3.3).

### 5.3.1 Trace loading optimizations

Invariant mining is a memory-bound application as the input trace involves millions of simulation instances and hundreds of variables. This makes, from the performance point of view, data-loading from the GPU main memory to the thread local registers one of the most important aspect of the whole mining procedure. *Mangrove* provides high throughput in the data loading phase by combining three different strategies: *double memory padding*, *vectorized memory accesses* combined with high thread-level data-parallelism, and *cache interleaving* data access.

The *double memory padding* strategy aims at efficiently organizing the input data into the GPU main memory in order to enable *memory coalescing*. The maximum memory coalescing is achieved when all threads of a warp access to continuous and *aligned* data in the device memory and, in particular, when data is aligned to the memory banks (of size 32 bytes for DRAM).

*Mangrove* organizes the input trace with a special function (`cudaMallocPitch`), which guarantees that all data elements of the input trace are aligned to the memory banks by adding padding at the end of each row of the trace. The data structure is also padded with an extra space in order to set the total number of columns as a multiple of the data loaded by a single step of a thread block. Such *ghost* columns and the corresponding speculative computation allow avoiding control flow statements to handle the mining of partial data at the trace borders.

Differently from the mining implementations at the state of the art, in which each thread loads a single trace value for each access until the whole trace is processed, *Mangrove* applies *vectorized memory accesses* to increase the size of the trace loaded by threads. GPU architectures support vectorized accesses to simplify load/store operations on aggregated built-in types (e.g., four integers, two chars, etc.). The same instructions (enabled via type-casting) allow accessing multiple data in a single memory access. This technique reduces the number of memory instructions four times for the most used data types (`int`, `unsigned`, and `float`).

The memory access bandwidth is further improved through *thread-level data parallelism*. It consists of performing many vectorized data loads per thread, while maintaining the correct stride among warp threads to preserve full memory coalescing. *Mangrove* implements such a feature through a local trace size (`L_SIZE`), which is set as a multiple of the vectorized access size and by unrolling the loops involved in the data loading. Loop unrolling allows avoiding the computation of loop conditions and loop iteration dependencies[3].

Finally, to reduce the massive use of the load/store units (LD/ST) in each SM, which may act as bottleneck of the entire application, *Mangrove* implements *cache interleaving*. Such a data-loading strategy exploits the *read-only memory*,

3. Loop unrolling is forced through the `#PRAGMA UNROLL` directive on the data load loops. Such a directive can be applied since the number of loop iterations is known at compile-time (see Section 5.3.3)

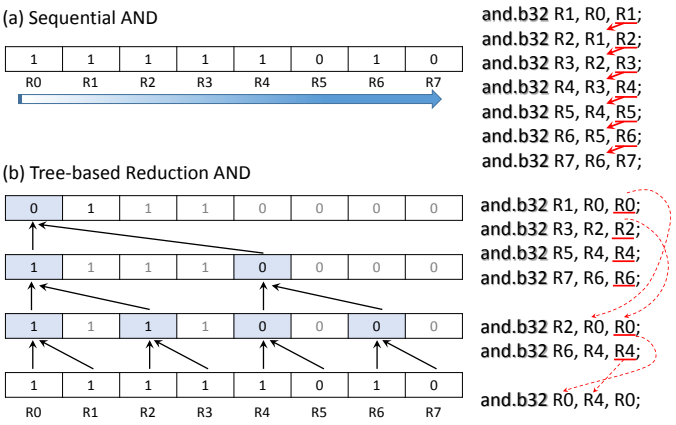


Fig. 6. Example of AND computation through the standard sequential approach (a) and through the tree-based reduction implemented in *Mangrove* (b).

which is provided by the most recent GPU architectures as a small cache for each SM. It implements a separate memory access pipeline and it is optimized for data that will not change during the kernel execution (e.g., the execution trace). The technique takes advantage of separate data-loading pipelines, by alternating standard memory accesses and read-only cache data-loading through CUDA `__ldg` instructions.

#### 5.3.2 Control-flow organization and thread coordination

*Mangrove* has been implemented to optimize the kernel instruction execution through *instruction-level parallelism* and through *atomic operations* during invariant checking. It ensures static indexing of registers by using unroll directive and C++ template parameters. This allows avoiding variable *spilling* in caches, reducing conditional statements, and breaking instruction dependencies.

One of the most time consuming tasks of the invariant checking is the computation of the final result by each thread (`templatesEval`, line 7 of Algorithm 2), which involves a heavy instruction serialization. It consists of performing the logic AND operation between an array element and its successor. Differently from the mining techniques at the state of the art that sequentially process each element, *Mangrove* implements a tree-based reduction to minimize the instruction dependencies and to maximize the instruction-level parallelism (ILP). Figures 6 shows an example of the standard sequential approach (a) and the optimized tree-based version implemented in *Mangrove* (b) (the representation of the sequential and parallel processes of the array in the left-most side, the corresponding low-level PTX instruction generated by the compiler in the right-most side). The instruction dependencies are highlighted in red. In the sequential process each instruction depends on the previous one, while in the tree-based reduction there are no dependencies among instructions of the same group.

The parallel implementation of a class of inference procedures (i.e., computing numeric ranges, as explained in Section 7.2) requires a wide use of atomic operations on floating-point data types. Even though GPU architectures support very efficient, yet hardware-implemented atomic operations on integer data types, they do not support atomic operations on floating-point data types. *Mangrove* implements a SW emulation of atomic operations on floating point through a mechanism based on an iterative compare-and-swap procedure. It relies on the standard IEEE754, which ensures that all float numbers are ordered lexicographically (i.e., given two positive numbers $x$ and $y$ with $x < y$, the bit representation of $x$ is always smaller than $y$ (opposite behaviour if negative numbers). This allows *Mangrove* to perform atomic comparisons between floating-point values by reinterpreting such values at bit-level as integers.

#### 5.3.3 GPU kernel auto-tuning

The kernel implementation for GPUs relies on templates (one per arity), whose parameters allow tuning:

- The size of thread blocks;
- The number of vectorized data load per thread (`L_SIZE`).

The right setting of these parameters allows achieving the best trade-off between (i) the maximum number of independent blocks, (ii) the minimum overhead for thread coordination, and (iii) the maximum device utilization.

The number of threads in a block mainly affects the synchronization overhead, the block scheduling efficiency, and the device occupancy. In general, smaller block sizes allow for a lightweight synchronization process, but they increase the number of blocks to be scheduled. On the other hand, larger blocks require a more expensive synchronization and they involve less overhead for the block scheduling. The minimum block size corresponds to the number of warp threads (i.e., 32 for CUDA), which allows completely avoiding intra-block synchronizations by adding conditional statements in the code (evaluated at compile-time) before the synchronization barriers. It enables the synchronization primitives only with block sizes greater than 32 threads.

The second parameter is the number of vectorized data-load per thread (`L_SIZE` / vector access size). Executing more than one independent data-load per thread improves the access concurrency and allows for a high instruction-level parallelism (ILP) in the subsequent computing phase. GPU threads are able to fully exploit all SM computation units (cores) and to achieve the full memory bandwidth by taking advantage of data-load and instruction-level parallelism. On the other hand, the number of vectorized data-load per thread heavily impacts on the number of registers used to store the thread local trace, which may compromise the device occupancy and, on border cases, it may reduce the application performance even with a high thread-level parallelism.

The parameter setting is *architecture-dependent* since it relies on the number of SMs, control logic, and memory characteristics of the GPU device. It is *data-independent* as, during the mining process, the number of variables and candidate invariants is static. For this reason, *Mangrove* implements an auto-tuning mechanism that executes the generated kernels over a small trace of random values (generated at run-time) on the target GPU device, and it sets the two kernel parameters for the complete mining phase.

In particular, the auto-tuning phase consists of executing a kernel with all the possible configurations of the kernel parameters in terms of thread block size (32, 64, 128, 256, 512, 1024, up to the maximum given by the device architecture) and thread level parallelism (ILP + vectrorized data load,

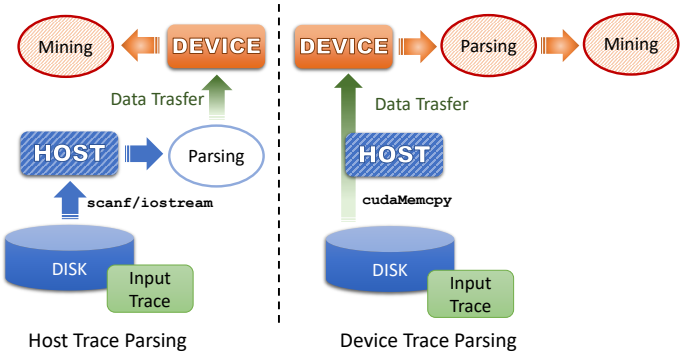


Fig. 7. Overview of host (i.e., standard approaches in literature) and device (i.e., implemented in *Mangrove*) trace parsing techniques.

ranging among 1, 2, 4, 8, 16)[4]. In our experimental results we obtained 30 different configurations. In general, all these values are stored in an internal configuration file, and the compiler uses such values to generate all the different kernels for a given GPU architecture. The tests are run on the user templates over a chunk of data trace. We claim (and we have been confirmed by the experimental results) that larger sizes of this chunk do not affect the best configuration choice.

## 6 Parsing execution traces from disk to GPU memory

The GPU kernel synthesized by *Mangrove* reduces the mining time from days/hours to few seconds in the worst case. This moves the bottleneck of the mining application from computation to the trace loading from disk. In the existing approaches, the input trace is read value by value using the standard API of C/C++ language (e.g., `scanf` or `iostream`), which can require several minutes to parse large input files (e.g., execution traces involving millions of time instants and hundreds of variables).

Differently from all the approaches in literature, *Mangrove* takes advantage of the massive parallelism of GPU devices to significantly reduce the time spent in the trace file reading. To the best of our knowledge, the proposed solution is the first that implements the input data parsing directly by the GPU device rather than by the CPU. The basic idea consists of copying the raw trace directly from the disk to the GPU memory (see Figure 7) by using the standard function `cudaMemcpy` combined with an invocation of specific kernels (see Sections 6.1 for boolean and 6.2 for numeric traces). After the parsing procedure, the execution trace is ready in the device memory for the mining process.

### 6.1 Parsing boolean traces

A boolean trace consists of a sequence of '0' or '1' characters (1-byte), each one followed by a single space. Different sequences of values associated to variables are organized in separated rows.

Following the same scheme of the mining process, *Mangrove* implements a GPU kernel to read boolean traces. It maps each block to a different row of the trace, thus guaranteeing independent thread execution. The starting row address is directly computed by multiplying the row id by two (a single boolean value requires exactly two bytes). In each block, the threads point to column indexes with a stride of 16 characters (8 boolean values) which correspond to the maximum size that can be loaded with a single instruction. The data is then converted into a bitmask of 1-byte.

The last step stores the parsed values into the GPU global memory. Instead of directly writing the bitmask values into the device memory, which can result in low memory throughput, the data is first copied into the shared memory. After the shared memory is completely filled, the data is organized into the device memory as explained in Section 5.3.1.

### 6.2 Numeric trace parsing

The raw numeric trace follows the structure of the boolean trace. Numeric values are separated by a whitespace and organized into rows, as shown in the example of Figure 8.

Parsing numeric traces has more issues than parsing boolean traces due to the irregular computation deriving from variable lengths of numeric values. For this reason, it is not possible to predict the starting position of each row and to directly map thread blocks to the corresponding addresses. The numeric parsing strategy consists of dividing the raw trace among GPU threads equally, and to deal with variable-length values by overlapping the loading of small memory chunks among adjacent thread blocks. Figure 9 shows an example of the whole parsing process, by considering two threads and five numeric values. The parsing procedure performs the following steps:

1) The starting memory address of a block is obtained by multiplying the block id by a fixed constant. Such value is computed as the amount of available shared memory per block minus the number of characters required to represent the longest numeric value (overlapped memory chunk).
2) Each block loads a contiguous sequence of trace characters into the shared memory. The number of loaded characters is equal to the available shared memory of a thread block.
3) The local trace stored in the shared memory is uniformly partitioned among block threads. Each thread sequentially reads its shared memory segment and converts the numeric values found. If a numeric value starts in a thread space and terminates in the next thread segment, the computation continues until the first whitespace is found. A numeric value is converted only if there is a whitespace before such number or by the first thread of a block.

The parsing algorithm extensively exploits the shared memory to avoid scattered device memory accesses and, consequently, poor performance. The small overlapped memory space loaded by adjacency blocks is essential to perform the computation in the shared memory even for *halo* elements among blocks.

At the end of the computation, each thread maintains its own converted values into local registers. At this step, the GPU threads cannot directly store such data into the device memory because each of them holds a different number of elements. *Mangrove* efficiently computes the thread output offsets without terminating the actual kernel by using an *online prefix-sum* procedure [19].

4. We set 16 as the maximum since it is generally the maximum value by considering the available registers. It can be easily increased.

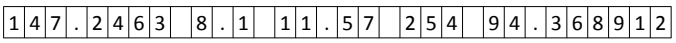


Fig. 8. Example of raw numeric trace.

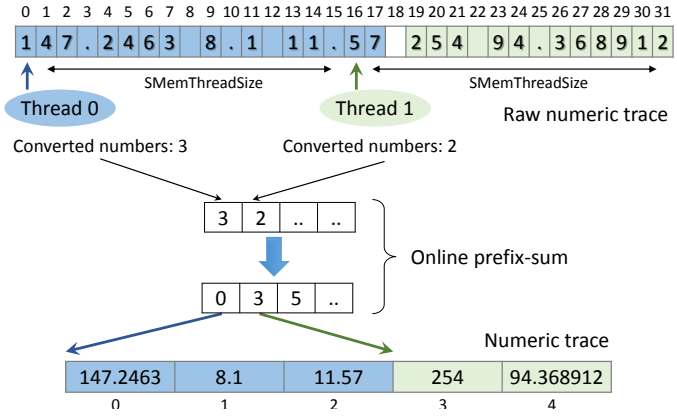


Fig. 9. GPU numeric trace parsing procedure.

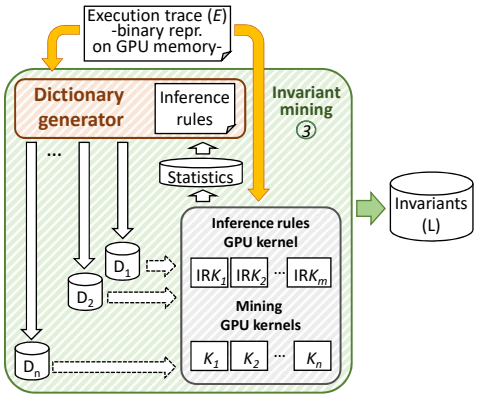


Fig. 10. Invariant miner of *Mangrove*.

The GPU trace parsing technique can be applied if the whole trace can be stored in the GPU memory. If this is not possible, *Mangrove* applies the standard technique based on reading and parsing through CPU. Switching from GPU to CPU parsing is statically set by considering the available GPU memory and the trace size ($trace\_size = \#\_of\_variables \times \#\_of\_instants \times 4$). In our tests, we successfully run the GPU parsing technique with traces with up to 5 millions of instants, 200 4-Bytes variables on a GPU device equipped with 3 GB. More recent GPU devices are generally equipped with more than 3 GB of memory. Considering that assertion mining on real problems is run after *data filtering* through logic cones analysis rather than on raw data, we claim that the proposed GPU-based parsing technique can be applied for mining real complex problems. In any case, for the mining of raw and very large traces, the framework can be switched on the CPU parsing modality.

## 7 Inference-based invariant mining

Figure 10 shows a more detailed overview of the invariant mining phase implemented in *Mangrove* (see Fig. 3). It starts from an execution trace $E$ and generates a set of invariants $L$.

The invariant miner generates one dictionary $\mathcal{D}_i$ for each Mining GPU kernel $\mathcal{K}_i$ ($\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_n$ in the example of Figure 10), where $\mathcal{D}_i$ collects all the possible permutations of $i$ variables. Each generated dictionary $\mathcal{D}_i$ and the simulation trace $E$ are then provided to the Mining GPU kernel $\mathcal{K}_i$, which extracts invariants to form $L$. This formulation of the invariant mining algorithm presents two major problems: scalability, and redundancy of the extracted invariants. In detail, generating each possible permutation of $i$ variables is infeasible for execution traces having hundreds of variables since the number of entries of $\mathcal{D}_i$ polynomially increases according to the formula $\prod_{k=0}^{i-1}(n-k)$, where $n$ is the number of variables and $i$ is the maximum arity. Furthermore, the generation of each possible permutation of $i$ variables can lead to redundant invariants. For instance, let $L = \{v_1 = false; v_2 == v_3\}$, then the invariant $v_1 == v_2 \oplus v_3$ is redundant as it does not provide any unknown relation among the three boolean variables $v_1$, $v_2$ and $v_3$. Indeed, it can be automatically inferred without using any Mining GPU kernel as $v_2 \oplus v_3$ is always false given $v_2 == v_3$. Consequently, the permutation $(v_1, v_2, v_3)$ can be omitted from $\mathcal{D}_3$ for the invariant template $bVar_1 == bVar_2 \oplus bVar_3$. Similarly, given the smallest $v_i^m$ and the largest $v_i^M$ value of a *numeric* variable $v_i$ in $E$, the invariant $v_i < v_j$ with $i \neq j$ and $v_i^M < v_j^m$ is redundant as it does not provide any unknown relation between the numeric variables $v_i$ and $v_j$. Therefore, all entries $(v_i, v_j)$ having $i \neq j$ and $v_i^M < v_j^m$ can be omitted from $\mathcal{D}_2$ for the invariant template $nVar_1 < nVar_2$.

With the aim of efficiently dealing with execution traces having hundreds of variables and reducing the number of extracted redundant invariants, *Mangrove* encodes some inference rules. Such rules, separately defined for *boolean* (Section 7.1) and *numeric* (Section 7.2) data types, are applied during the generation of the dictionary $\mathcal{D}_i$. The inference rules rely on input data characteristics that can be inferred from $E$, such as: the largest value of a *numeric* variable, the number of simulation instants of $E$ in which a candidate invariant holds, etc. It is worth noting that all Mining GPU kernels $\mathcal{K}$ can only be applied to mine invariants from $E$. Consequently, some Inference Rule GPU kernels (IRK) are also encoded in *Mangrove* to provide the introduced inference rules with the required statistics.

### 7.1 Boolean inference

For *boolean* data type, the inference rules applied during the generation of a dictionary $\mathcal{D}_i$ rely on the *occurrence* of an invariant instantiation. Let *inv* be the instantiation of an invariant template, the *occurrence* of *inv* ($|inv|$) is the number of simulation instants of $E$ in which *inv* holds. In order to compute the *occurrence* of an *inv*, the Inference Rule GPU kernels, $IRK_1$ and $IRK_2$, were encoded in *Mangrove*. Given a dictionary $\mathcal{D}_1$, the kernel $IRK_1$ computes each *occurrence* of $v_i == true$ where $v_i$ is an entry of $\mathcal{D}_1$. Similarly, given a dictionary $\mathcal{D}_2$, the kernel $IRK_2$ computes each *occurrence* of $v_i == v_j$ where $(v_i, v_j)$ is an entry of $\mathcal{D}_2$.

The execution flow of the invariant miner consists of the following three sequential steps:

**(1)** The invariant miner extracts the invariants involving only a *boolean* variable, namely all the *boolean* variables holding

the same value in all instants $\mathcal{T}$ of the trace $E$. The invariant miner generates the dictionary $\mathcal{D}_1 = \{v_i | v_i \in \mathcal{B}\}$. Then, the Inference Rule GPU kernel $IRK_1$ is run. Finally, *Mangrove* extracts the invariant $v_i == true$ if $|v_i == true|$ is equal to $|E|$, namely the number of simulation instants in $E$. Similarly, it extracts $v_i == false$ if $|v_i == true|$ is equal to zero.

**(2)** The invariant miner extracts all the invariants involving two *boolean* variables. According to the invariant template grammar introduced in Section 5.1, only the invariant templates $bTerm == bTerm$ and $bTerm == !bTerm$ can be defined. Therefore, the invariant miner generates the dictionary $\mathcal{D}_2 = \{(v_i, v_j) | v_i, v_j \in \mathcal{B}; i < j; 0 < |v_i| < |E|; 0 < |v_j| < |E|\}$. The constraint $i < j$ prevents both the permutations $(v_i, v_j)$ and $(v_j, v_i)$ from being added in $\mathcal{D}_2$. $0 < |v_i| < |E|$ and $0 < |v_j| < |E|$ prevent the invariant miner from generating a permutation involving variables always holding the same value since the equality and inequality relation between constants is redundant. Then, the Inference Rule GPU kernel $IRK_2$ is applied to get $|v_i == v_j|$ for each entry of $\mathcal{D}_2$. Finally, *Mangrove* extracts the invariant $v_i == v_j$ if $|v_i == v_j| = |v_i|$, while it extracts $v_i = !v_j$ if $|v_i == v_j|$ is equal to zero.

**(3)** as third and last step, the invariant miner extracts the invariants involving three *boolean* variables. According to the invariant template grammar introduced in Section 5.1, only the invariant templates meeting $bTerm == bTerm\, bOperator\, bTerm$ can be defined. In this case, the dictionary generator creates the dictionary $\mathcal{D}_3 = \{(v_i, v_j, v_k) | v_i, v_j, v_k \in \mathcal{B}; i \neq j \neq k; j < k; 0 < |v_i| < |E|; 0 < |v_j| < |E|; 0 < |v_k| < |E|; |v_i| == |v_j\, bOperator\, v_k|\}$. In detail, $0 < |v_i| < |E|$, $0 < |v_j| < |E|$ and $0 < |v_k| < |E|$ prevent the invariant miner from generating a permutation involving variables always holding the same value. The constraint $j < k$ prevents both the permutations $(v_i, v_j, v_k)$ and $(v_i, v_k, v_j)$ from being added in $\mathcal{D}_3$ as any *bOperator* has the commutative property. Finally, $|v_i| == |v_j\, bOperator\, v_k|$ introduces an existence constraint for any ternary invariant template. Figure 11 shows the decomposition rules introduced in *Mangrove* to apply the introduced existence constraint with any user-defined invariant template. As an example, consider the invariant instantiation $v_i == \neg v_j \vee v_k$. This instantiation can exists in $E$ as long as $|v_i|$ is equal to $|\neg v_j \vee v_k|$. Through the decomposition rules, we can rewrite this constraint as $|v_i| == |\neg v_j| + |v_k| - |!v_j \wedge v_k|$, and finally as $|v_i| == |E| - 2 * |v_j| + |v_k| + |v_j \wedge v_k|$.

By exploiting the statistics generated in the first and second step the defined existence constraint is successfully applied to remove permutations of variables from $\mathcal{D}_3$ that cannot satisfy any ternary invariant template.

### 7.2 Numeric inference

For *numeric* data type, the inference rules applied during the generation of a dictionary $\mathcal{D}_i$ rely on the smallest $(v_i^m)$ and largest $(v_i^M)$ value of a *numeric* variable $v_i$, and the monotonic/commutative property of invariant templates. Given an invariant template $it$ : $nVar\ nRelation\ rExpr$, $it$ is monotonic if $rExpr$ is either monotonically increasing or decreasing:

$$\frac{|\neg v_i|}{|E| - |v_i|} \qquad \frac{|v_i \vee v_j|}{|v_i| + |v_j| - |v_i \wedge v_j|}$$
$$\frac{|\neg v_i \wedge v_j|}{|v_i| - |v_i \wedge v_j|} \qquad \frac{|v_i \oplus v_j|}{|\neg v_i \wedge v_j| + |v_i \wedge \neg v_j|}$$

Fig. 11. Decomposition rules for boolean invariant templates.

- For each $c, \Delta \in \mathbb{R}$ with $\Delta \geq 0$, a 3-arity invariant template $it$ is *monotonically increasing* if $rExpr(c + \Delta) - rExpr(c) \geq 0$. Similarly, $rExpr$ is *monotonically decreasing* if $rExpr(c + \Delta) - rExpr(c) \leq 0$.
- For each $c_1, c_2, \Delta \in \mathbb{R}$ with $\Delta \geq 0$, a 3-arity invariant template $it$ is *monotonically increasing* if $rExpr(c_1 + \Delta, c_2) - rExpr(c_1, c_2) \geq 0$ and $rExpr(c_1, c_2 + \Delta) - rExpr(c_1, c_2) \geq 0$. Similarly, $rExpr$ is *monotonically decreasing* if $rExpr(c_1 + \Delta, c_2) - rExpr(c_1, c_2) \leq 0$ and $rExpr(c_1, c_2 + \Delta) - rExpr(c_1, c_2) \leq 0$.

For each $c_1, c_2 \in \mathbb{R}$, a 3-arity invariant template $it$ is *commutative*, if $rExpr(c_1, c_2) - rExpr(c_2, c_1) == 0$.

The Inference Rule GPU kernel $IRK_1$ allows computing the smallest and largest value of a *numeric* variable $v_i$. Given a dictionary $\mathcal{D}_1$, the kernel $IRK_1$ computes the smallest $v_i^m$ and largest $v_i^M$ of each entry $(v_i) \in \mathcal{D}_1$.

If we do not have all monotony and commutative templates, *Mangrove* checks each single 2 and 3 variable permutations with each user-defined template. Otherwise, the execution flow of the dictionary generator for monotonic and commutative invariant templates implements the following three sequential steps:

**(1)** The invariant miner extracts the invariants involving only a *numeric* variable, namely all the *numeric* variables always holding the same value in the whole trace $E$. In detail, the invariant miner generates the dictionary $\mathcal{D}_1 = \{v_i | v_i \in \mathcal{N}\}$. Afterwards, the Inference Rule GPU kernel $IRK_1$ is applied. Finally, *Mangrove* extracts the invariant $v_i == const$ with $const = v_i^m$ if $v_i^m == v_i^M$.

**(2)** as second step, the invariant miner extracts all the invariants involving two *numeric* variables. According to the invariant template grammar introduced in Section 5.1, only the invariant template $nVar_1\ nRelation\ rExpr(nVar_2)$ can be defined. The invariant miner generates the dictionary $\mathcal{D}_2 = \{(v_i, v_j) | v_i, v_j \in \mathcal{N}; i \neq j; C\left(nRelation, rExpr, v_i^m, v_i^M, v_j^m, v_j^M\right)\}$. The constraint $C\left(nRelation, rExpr, v_i^m, v_i^M, v_j^m, v_j^M\right)$ prevents the permutation $(v_i, v_j)$ from being added in $\mathcal{D}_2$ according to the $nRelation$ operator. In detail, $C$ is defined as follows:

$$C\left(nRelation, rExpr, v_i^m, v_i^M, v_j^m, v_j^M\right) =$$

$$\begin{cases} v_i^m == \min\left(rExpr\left(v_j^m\right), rExpr\left(v_j^M\right)\right) and \\ v_i^M == \max\left(rExpr\left(v_j^m\right), rExpr\left(v_j^M\right)\right) & \text{if } rRelation \text{ is } == \\ \\ v_i^M > \min\left(rExpr\left(v_j^m\right), rExpr\left(v_j^M\right)\right) or \\ v_i^m < \max\left(rExpr\left(v_j^m\right), rExpr\left(v_j^M\right)\right) & \text{if } rRelation \text{ is } <, \neq \\ \\ v_i^M \geq \min\left(rExpr\left(v_j^m\right), rExpr\left(v_j^M\right)\right) or \\ v_i^m \leq \max\left(rExpr\left(v_j^m\right), rExpr\left(v_j^M\right)\right) & \text{if } rRelation \text{ is } \leq \end{cases}$$

If *nRelation* is the equality operator, then $v_i^m$ and $v_i^M$ have to respectively be equal to the smallest and largest value computed by *rExpr* with $v_j$ as input parameter. Because of the monotony property of *rExpr*, the smallest and largest value are respectively $min\big(rExpr(v_j^m), rExpr(v_j^M)\big)$ and $max\big(rExpr(v_j^m), rExpr(v_j^M)\big)$. Similarly, when *nRelation* is not the equality operator, a candidate invariant is evaluated on condition that at least a value of $v_i$ is in $[rExp(v_j^m), rExp(v_j^M)]$.

Figure 12 shows an example where the invariant $v_1 < v_2+7$ can directly be inferred without using any mining kernel. In this example, the variable $v_1$ has the smallest and largest value respectively equals to 2 and 5. Meanwhile, the variable $v_2$ has the smallest and largest value respectively equals to 1 and 8. Because *rExpr* is monotonically increasing, $rExpr(v_2)$ has to compute values within the interval $[8, 15]$. Since $v_1^M$ is smaller than $min(rExpr(v_2^m), rExpr(v_2^M))$ (namely $5 < 8$), then no counter example can exist in $E$ to falsify $v_1 < v_2 + 7$. Consequently, the entry $(v_1, v_2)$ can be omitted from $\mathcal{D}_2$.

**(3)** as third and last step, the invariant miner extracts all the invariants involving three *numeric* variables. According to the invariant template grammar introduced in Section 5.1, only the invariant template $nVar_1\, nRelation\, rExpr(nVar_2, nVar_3)$ can be defined. The invariant miner generates the dictionary $\mathcal{D}_3 = \{(v_i, v_j, v_k) | v_i, v_j, v_k \in \mathcal{N}; i \neq j \neq k; j < k; C'\left(nRelation, rExpr, v_i^m, v_i^M, v_j^m, v_j^M, v_k^m, v_k^M\right)\}$. The constraint $j < k$ prevents both the permutations $(v_i, v_j, v_k)$ and $(v_i, v_k, v_j)$ from being added in $\mathcal{D}_3$ as *rExpr* has the commutative property. The constraint $C'\left(nRelation, rExpr, v_i^m, v_i^M, v_j^m, v_j^M, v_k^m, v_k^M\right)$ prevents the permutation $(v_i, v_j, v_k)$ from being added in $\mathcal{D}_3$ according to the *nRelation* operator. In detail, $C'$ is defined as follow:

$$C'\left(nRelation, rExpr, v_i^m, v_i^M, v_j^m, v_j^M, v_k^m, v_k^M\right) =$$

$$\begin{cases} v_i^m == \min\left(rExpr\left(v_j^m, v_k^m\right), rExpr\left(v_j^M, v_k^M\right)\right) \text{ and} \\ v_i^M == \max\left(rExpr\left(v_j^m, v_k^m\right), rExpr\left(v_j^M, v_k^M\right)\right) & \text{if } rRelation \text{ is } == \\ v_i^M > \min\left(rExpr\left(v_j^m, v_k^m\right), rExpr\left(v_j^M, v_k^M\right)\right) \text{ or} \\ v_i^m < \max\left(rExpr\left(v_j^m, v_k^m\right), rExpr\left(v_j^M, v_k^M\right)\right) & \text{if } rRelation \text{ is } <, \neq \\ v_i^M \geq \min\left(rExpr\left(v_j^m, v_k^m\right), rExpr\left(v_j^M, v_k^M\right)\right) \text{ or} \\ v_i^m \leq \max\left(rExpr\left(v_j^m, v_k^m\right), rExpr\left(v_j^M, v_k^M\right)\right) & \text{if } rRelation \text{ is } \leq \end{cases}$$

As for the constraint $C$ introduced in the second step, $C'$ relies on the monotony property of *rExpr* to evaluate a candidate invariant. In detail, $v_i^m$ and $v_i^M$ have to match respectively the smallest and largest value computed by *rExpr* with $v_j$ for the equality operator. If *nRelation* is different from the equality operator, then at least a value of $v_i$ has to be in $[rExp(v_j^m, v_k^M), rExp(v_j^M, v_k^M)]$.

## 8 Experimental results

We evaluated the efficiency of *Mangrove* and of the mining techniques presented in this article on a set of representative IP-core benchmarks from the *OpenCores* library [20].We considered an implementation of the AMBA-BUS protocol apb-bus [21], a floating-point adder fp-add, a floating-point multiplier fp-mul, an asynchronous receiver/transmitter uart,

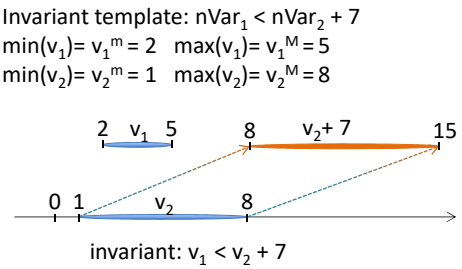


Fig. 12. A 2-arity invariant for the pair of numeric variables $a$ and $b$

| Boolean | Numeric |
|---|---|
| $u = v$ | $u = v$ |
| $u = \neg v$ | $u \neq v$ |
| $u = v \wedge w$ | $u < v$ |
| $u = v \vee w$ | $u \leq v$ |
| $u = v \oplus w$ | $u = min(v, w)$ |
| $u = v \rightarrow w$ | $u = max(v, w)$ |
| $u = v \leftarrow w$ | $u = v * w$ |
| $u = \neg(v \wedge w)$ | $u = v + w$ |
| $u = \neg(v \vee w)$ | |
| $u = \neg(v \oplus w)$ | |
| $u = \neg(v \rightarrow w)$ | |
| $u = \neg(v \leftarrow w)$ | |

TABLE 2
*Set of invariant templates used in the experimental results.*

and a jpeg image compression algorithm. For each of them, we extracted an execution trace through simulation, either by applying random stimuli or by generating interface-compliant stimuli as proposed in [22]. We considered both internal and interface signals. For the jpeg IP, we considered each subcomponent individually. We then split each execution trace in two traces containing either the *boolean* or the *numeric* variables of the design, respectively. Sections 8.2 and 8.1 present the analysis of the obtained results. We run the experiments on an AMD Phenom II X6 1055T processor equipped with 8GB of memory, Ubuntu 14.04 OS, and connected to an NVIDIA GeForce GTX 780 GPU consisting of 12 SMs with 2,304 CUDA cores, and CUDA Toolkit 9.0.

### 8.1 *Mangrove* analysis and comparison with the state of the art approaches

We evaluated and compared *Mangrove* with *Daikon* [13] and *Turbo* [16], which are respectively the most representative sequential and parallel implementations at the state of the art.

*Daikon* is one of the most flexible and effective sequential tool for invariant mining [18]. *Turbo* is, for the best of our knowledge, the only existing parallel implementation for GPUs. To perform a fair comparison, since Mangrove supports a superset of templates with respect to Daikon and Turbo, we defined a set of templates compatible with all tools (Table 2)[5]. The templates include *boolean* and *numeric* relations with different arity and functions to deeply stress the inference algorithm and the mining procedure. We performed the functional validation of the results by checking that the

5. It is important to note that the state of the art approaches either do not provide support for custom templates (e.g., *Turbo*) or they provide a limited support for some specific templates (e.g., *Daikon*).

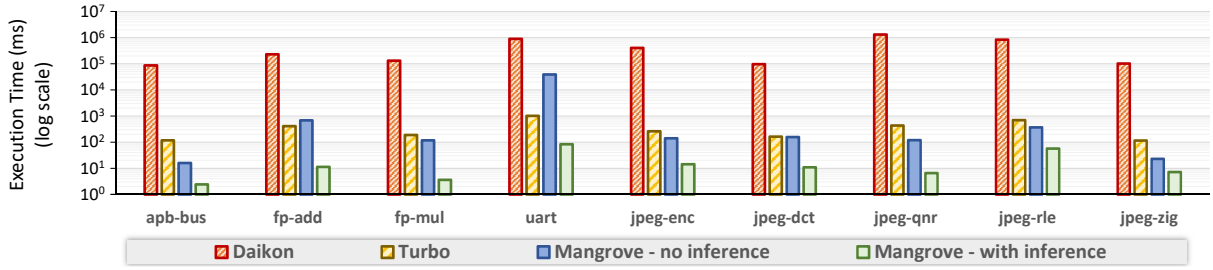


Fig. 13. Performance comparison of Mangrove with the most representative implementation at the state of the art for boolean traces (simulation instances 20,000,000).

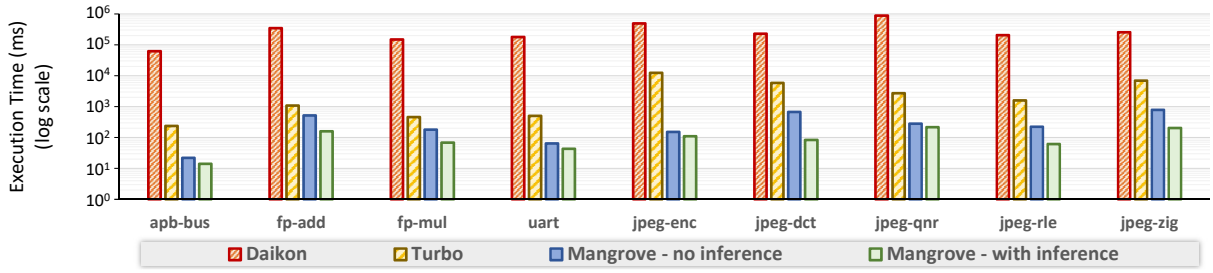


Fig. 14. Performance comparison of Mangrove with the most representative implementation at the state of the art for numeric traces (simulation instances 5,000,000).

three tools (*Daikon*, *Turbo*, and *Mangrove*) return exactly the same set of invariants. This was expected, since there is no difference among the three tools concerning their ability of extracting the invariants, given the same set of templates. The only difference, except the higher flexibility of Mangrove that allows the users defining a larger set of customized templates, is related to the efficiency of the mining, which is analysed in the following paragraphs.

Figure 13 and 14 report the results in terms of execution time (in ms, logarithmic scale) required by the tools to mine invariants from *boolean* and *numeric* execution traces. respectively. For *Mangrove*, the figures report the execution time by disabling and enabling the inference support. The traces consist of 20 and 5 millions of simulation instants, respectively. The results show that *Mangrove* is up to five orders of magnitude faster than *Daikon* and two orders of magnitude faster with respect to *Turbo*. This is mainly do to the fact that *Mangrove* efficiently exploits inference to reduce the problem complexity (see Section 7). *Mangrove* outperforms the other tools even when inference is not enabled and this is due to the several optimizations implemented to fully exploit the GPU architectural characteristics (see Section 5.3). Only in the case of *boolean* traces with a very large number of invariants (e.g., uart), *Turbo* is faster than *Mangrove* when inference is not enabled. This is due to the fact that *Turbo* implements basic inference mechanisms, which help avoiding computation for several redundant relations and which could not be disabled for our comparison.

In general, the inference rules implemented in *Mangrove* allow reducing the mining time up to two orders of magnitude. The number of dictionary entries removed through the inference rules depends on the benchmark and on the type of execution trace as explained in the next session.

We also found that CUDA *multi-streams* are unlikely to improve the overall performance since: (1) the GPU is already fully utilized during the mining phase. (2) Even splitting the traces in chunks, the contribution of multi-streams to the overall performance is limited as the time spent in memory transfer is negligible compared to the kernel execution. (3) Multi-streams and pipelining can be applied only if CPU the trace parsing is performed with the CPU. Considering that GPU trace parsing can lead up to 100x speedup and, for what claimed before pipelining cannot provide such a speedup, multi-stream+pipeling is worth to be applied only with CPU parsing. In addition, as shown by the experimental results, the benefits given by the intensive GPU register usage is strongly related to thread/data parallelism. Finally, the auto-tuning technique implemented in the framework aims at improving the trade-off between register usage and Instruction-level parallelism. *Mangrove* relies on the register usage to increase the vectorized data-load per thread, while it does not exceed their usage to compromise the maximum device occupancy (as underlined in Section 5.3.3).

### 8.1.1 *Inference Efficiency Analysis*

Tables 3 and 4 show how *Mangrove* can reduce the problem complexity through inference for the *boolean* and the *numeric* traces, respectively. For each benchmark, the table report the number of variables, the number of extracted invariants ($\mathcal{L}$), the dictionary size ($\mathcal{D}$) by enabling or disabling the inference rules, and the percentage of entries removed from the dictionary through inference. In general, the results show that the inference rules allow reducing the dictionary size up to 99.5% and that they are more effective in case of *boolean* traces. This is due to the fact that the number of value combinations in boolean traces is limited compared to *numeric* traces and, as a consequence, it is easier to discard relations that do not satisfy the extracted values.

The inference rules significantly improve the overall performance in two ways. First, they allow saving a considerable amount of time as the complexity of the mining algorithm strictly depends on the dictionary size. Such a relationship is

| Benchmark | Vars | Invariants | Dictionary Size | | Removed by inference |
|---|---|---|---|---|---|
| | | | No inference | With inference | |
| apb-bus | 24 | 1,150 | 6,348 | 199 | 96.9% |
| fp-add | 60 | 68,440 | 104,430 | 1,056 | 99.0% |
| fp-mul | 36 | 11,172 | 22,050 | 459 | 97.9% |
| uart | 169 | 5,530,972 | 2,384,928 | 12,904 | 99.5% |
| jpeg-enc | 44 | 14,264 | 40,678 | 9,227 | 77.3% |
| jpeg-dct | 156 | 18,480 | 30,360 | 4,989 | 83.6% |
| jpeg-qnr | 42 | 14,326 | 861 | 465 | 97.4% |
| jpeg-rle | 78 | 35,301 | 231,231 | 23,366 | 89.9% |
| jpeg-zig | 26 | 1,692 | 8,125 | 1,927 | 76.3% |

TABLE 3
*Boolean benchmark dataset. The table reports the benchmark characteristics and the corresponding number of dictionary entries for boolean traces with/without inference.*

| Benchmark | Vars | Invariants | Dictionary Size | | Removed by inference |
|---|---|---|---|---|---|
| | | | No inference | With inference | |
| apb-bus | 16 | 168 | 3,600 | 1,323 | 63.3% |
| fp-add | 56 | 3,346 | 169,400 | 25,755 | 84.8% |
| fp-mul | 30 | 1,304 | 25,230 | 3,842 | 84.8% |
| uart | 30 | 349 | 14,400 | 3,640 | 74.7% |
| jpeg-enc | 109 | 546 | 1,236,492 | 452,143 | 63.4% |
| jpeg-dct | 40 | 2,924 | 3,747,900 | 343,048 | 90.7% |
| jpeg-qnr | 76 | 1,923 | 188,442 | 56,938 | 69.8% |
| jpeg-rle | 77 | 1,228 | 394,752 | 52,948 | 86.6% |
| jpeg-zig | 204 | 1,569 | 8,283,212 | 391,528 | 95.3% |

TABLE 4
*Numeric benchmark dataset. The table reports the benchmark characteristics and the corresponding number of dictionary entries for numeric traces with/without inference.*

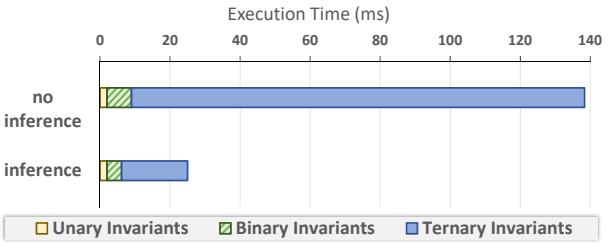


Fig. 15. Workload breakdown for boolean mining. Benchmark jpeg-enc, simulation instances 10,000,000.

better underlined in Figure 15, which reports, for one of the considered benchmarks ($jpeg-enc$), the execution time spent for mining unary, binary, and ternary entries of the dictionary. Without inference, the ternary relations require much more time due to the number of cases that have to be verified. With inference, the execution time is drastically reduced. Second, the inference rules allow the tool to avoid transferring most of the dictionary entries from the disk to the GPU memory.

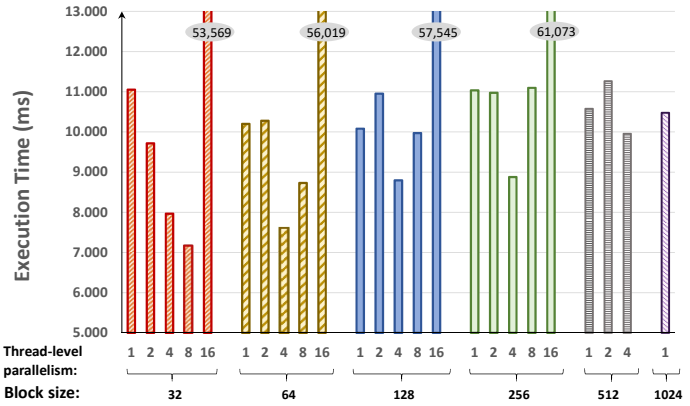


Fig. 16. Mangrove performance for different tuning parameters (block size and thread-level parallelism). Simulation instances 500,000, number of variables: 80, number of dictionary entries: 492,960, template: numeric ternary.

### 8.1.2 *Effectiveness of the tuning mechanism*

The results discussed in the previous sections have been obtained with the best tool configuration obtained with the *auto-tuning* mechanism presented in Section 5.3.3. To show the importance of such a mechanism, Figure 16 reports the mining time obtained with different manual configurations, each one set by varying the thread block size and the thread-level parallelism. The configurations not reported can not be compiled due to the limitation of the number of available registers in the GPU device.

The fine tuning of *Mangrove* plays an essential role to achieve the full mining efficiency, while a wrong configuration can lead to poor performance. Small block sizes help reducing the execution time thanks to the low synchronization overhead, while the thread-level parallelism is effective when the kernel configuration does not allow the full occupancy of the device (i.e. block size of 32/64 threads). We observed a significant performance decreasing for a value of thread-level parallelism equal to 16 due to the high register *spilling* in cache L1 required to maintain the local trace in the thread memory space.

In general, such a auto-tuning mechanism represents an important contribution to improve the overall approach scalability and portability to different GPU architectures. In our architecture, the speedup from the fastest configuration (block size 32, thread-level parallelism 8) and the slowest one (block size 256, thread-level parallelism 16) is 8.7x.

## 8.2 Trace parsing evaluation

Reading and parsing the execution traces play a key role in the overall approach efficiency. While the GPU mining kernel requires only few seconds to analyse traces with hundreds of variables and millions of simulation instants (see Section 8.1), loading and parsing such traces with standard approaches may require much more effort. Table 5 illustrates a comparison between the standard (CPU-based) technique and the proposed GPU-based technique (see Section 6). The table reports the trace type (*boolean* or *numeric*), the trace size (in MB), the data parsing time, the data transfer time between host and device, and the total time (i.e., the sum of parsing and transfer time). The last column underlines the speed-up.

| Trace type | Trace size | CPU | | | GPU | | | Total speedup |
|---|---|---|---|---|---|---|---|---|
| | | Parsing (ms) | Data Transfer to GPU (ms) | Total (ms) | Parsing (ms) | Data Transfer to GPU (ms) | Total (ms) | |
| **Boolean** | 1,052 MB | 11,091.1 | 19.0 | 11,110.1 | 5.4 | 375.0 | 380.4 | **29.2x** |
| **Numeric** | 1,000 MB | 42,241.0 | 142.0 | 42,383.0 | 21.8 | 355.8 | 377.6 | **112.2x** |

TABLE 5
*Performance comparison between CPU and GPU data transfer and parsing techniques.*

The results underline that the parsing data phase is the bottleneck in the standard CPU-based approach, while the data transfer time is negligible. They also confirm that, due to their more complex representation, *numeric* traces require more parsing time (four times) than *boolean* traces. On the other hand, the data transfer time is much more time consuming in the proposed GPU-based approach. However, the proposed kernel to parsing data on the GPU is up to 2,000× faster than the standard CPU approach for both *numeric* and *boolean* traces. This makes the proposed solution, in total, 29.2× faster for *boolean* traces and 112.2× faster for *numeric* traces then the standard approach. We also observed that the bitmask representation for the boolean trace translates into a very fast data transfer for the CPU procedure, while it does not affect the GPU solution due to the raw trace transfer (see section 6.1 and 6.2).

The optimizations described in sections 5.3.1 and 5.3.2 are embedded in the algorithm implemented by *Mangrove* and cannot be selectively switched off. On the other hand, for the optimizations evaluated in Sections 6.1 and 6.2, we compared *Mangrove* and the other state-of-the-art approaches without including the parsing time (see Fig. 14). Table 5 reports the parsing time for host and GPU.

## 9 Conclusion and future work

This article presented *Mangrove*, an efficient implementation of the dynamic invariant mining algorithm for GPU architectures. The proposed solution starts from user-defined invariant templates and execution traces of a system under verification, and it generates a library of likely invariants. The tools exploits inference rules to sensibly reduce the mining effort and has been implemented for the parallel execution on GPU architectures. The article presented the analysis of the tool performance on a set of widely-used IP models. Compared to the representative solutions at the state of the art, the tool allows sensibly reducing the execution time up to one and two orders of magnitude with respect to the best parallel and sequential solutions, respectively, at the state of the art.

## References


[1] V. Braberman, D. Garbervetsky, and S. Yovine, "A static analysis for synthesizing parametric specifications of dynamic memory consumption," *J. of Object Technology*, vol. 5, no. 5, pp. 31–58, 2006.

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

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

[4] Y.-R. Chen, J.-J. Yeh, P.-A. Hsiung, and S.-J. Chen, "Accelerating coverage estimation through partial model checking," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1613–1625, 2014.

[5] 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*, 1997, pp. 391–411.

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

[7] A. Danese, N. Riva, and G. Pravadelli, "A-team: Automatic template-based assertion miner," in *Proc. of ACM/IEEE Design Automation Conference (DAC)*, vol. Part 128280, 2017.

[8] R. Vemu and J. Abraham, "Ceda: Control-flow error detection using assertions," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1233–1245, 2011.

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

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

[11] C. Flanagan, R. Joshi, and K. R. M. Leino, "Annotation inference for modular checkers," *Inf. Process. Lett.*, vol. 77, no. 2-4, pp. 97–108, 2001.

[12] N. Tillmann, F. Chen, and W. Schulte, "Discovering likely method specifications," in *Formal Methods and Software Engineering*, ser. LNCS, Z. Liu and J. He, Eds. Springer, 2006, vol. 4260, pp. 717–736.

[13] 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*, vol. 69, no. 1, pp. 35–45, 2007.

[14] 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*, 2005, pp. 775–778.

[15] 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*, 2006, pp. 169–180.

[16] A. Danese, L. Piccolboni, and G. Pravadelli, "A parallelizable approach for mining likely invariants," in *Proc. of ACM/IEEE CODES+ISSS*, 2015.

[17] N. Bombieri, F. Busato, A. Danese, L. Piccolboni, and G. Pravadelli, "Exploiting gpu architectures for dynamic invariant mining," in *Proc. of IEEE International Conference on Computer Design, ICCD 2015*, 2015, pp. 192–195.

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

[19] S. Yan, G. Long, and Y. Zhang, "Streamscan: fast scan algorithms for gpus without global barrier synchronization," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 229–238.

[20] [Online]. Available: http://opencores.org/

[21] [Online]. Available: http://www.arm.com/products/system-ip/amba-specifications

[22] L. Piccolboni and G. Pravadelli, "Simplified stimuli generation for scenario and assertion based verification," in *Proc. of IEEE Latin American Test Workshop (LATW)*, 2014, pp. 1–6.

**Nicola Bombieri** is Associate Professor at the Department of Computer Science, University of Verona, Italy. His research activity focuses on parallel and heterogeneous architectures, parallel computing, and parallel programming languages. He develops techniques to efficiently parallelize software applications for multi-core, many-core, heterogeneous architectures targeting performance, power, and energy efficiency. His research field also includes electronic design automation (EDA) applied to Smart Systems modeling and verification, automatic generation of embedded parallel software, hardware description languages (HDLs), EDA applied to Systems Biology for network modeling and simulation. He serves as Technical Program Committee member, Program Chair, Workshops/Special sessions Chair at ACM/IEEE conferences like DAC, DATE, ICCD, MCSoC, SIES, ECSI FDL, CODES/ISSS, MEMOCODE, DSD, VLSI-SoC, ETS. He is author of more than 100 publications in international journals and conferences. He is Editor of two books.

**Federico Busato** received the Ph.D. in Computer Science from the University of Verona in 2018. Currently, he is a senior software engineer at Nvidia. His research activity focuses on high-performance computing, graph analytics, and sparse linear algebra.

**Alessandro Danese** received the Master degree and the PhD in Computer Science from the University of Verona, Italy, respectively, in 2014 and 2018. He has been also visiting student at the University of Michigan (USA) in 2016 and at Intel Corporation (USA) in 2017. His main research interests are related to assertion-based verification and they focus particularly on assertion mining and assertion coverage. In this context, he collaborated in scientific and industrial projects. He is a member of the IEEE.

**Luca Piccolboni** (S'15) received the B.S. degree (summa cum laude) in Computer Science from the University of Verona, Verona, Italy, in 2013, and the M.S. degree in Computer Science and Engineering (summa cum laude) from the University of Verona in 2015. He is currently working toward the Ph.D. degree in Computer Science at Columbia University, New York, NY, USA. His research interests include design and verification of embedded systems, with particular regard to computer-aided design, high-level synthesis, hardware acceleration, and system security.

**Graziano Pravadelli** , PhD in computer science, IEEE senior member, IFIP 10.5WG member, is full professor of information processing systems at the Computer Science Department of the University of Verona (Italy) since 2018. In 2007 he co-founded EDALab s.r.l., an Italian SME whose mission consists of giving support for the innovation and technology transfer in embedded system modeling and verification. His main interests focus on system-level modeling, simulation and semi-formal verification of HW/SW embedded systems and cyber physical systems More recently he started to study the application of embedded systems to develop virtual coaching platforms for people with special needs. In the previous contexts, he collaborated in several national and European projects and he published more than 100 papers in international conferences and journals.