Improving Performance with IEV

By Jörg Müller, Cadence Design Systems, September 2012

Version 1.0,

Referring to Incisive Enterprise Verifier 12.1

Overview

Formal verification is much more depending on complexity than simulation. As designs to verify get bigger formal is subject to unappealing results like “Explored” which means “I can’t give you a guarantee” (which is the answer you get from a simulation by definition, by the way). Therefore it is very important to control and manage the complexity in order to achieve the best possible performance and improve the chance to get the desired guarantees.

There are 3 main ingredients to performance:

1. Complexity of the DUT

   The DUT contributes complexity in form of counters, memories, FIFOs and other combinational logic. The main methods to reduce it are parameterization and abstraction (replacement) of particularly complex constructs.

2. Complexity of the Environment

   The assertion to check, the set of constraints and auxiliary logic in the fan-in cone of the assertion make up the complexity introduced by the environment. Splitting or reducing the scope of the assertion has a big impact on performance. Users do also disable or add constraints in an attempt to create verification cases with reduced complexity.

3. Formal Engines

   The third ingredient to performance is the formal engines. The user can select a set of formal engines, orchestrate their operation, and tune the model that enters the engines for efficient processing.

This document describes some key features of IEV that impact performance of the formal engines. It assumes that all the pre-checking analysis to clean up the constraints and optimize the environment is already performed.

Invocation

The easiest way to optimize performance is to invoke the tool as follows:

```
iev +turbo ...
```

This will apply a series of other invocation commands and runtime settings, some of which are described below, to get you the most out of the box performance for competitive analysis.

This is the first thing to try if you need to improve performance and run on a multi-CPU machine. More fine grained controls are described in the following chapters.

Effort

The most important setting is the effort. The formal engines run until they receive a user defined timeout as specified by this effort setting:

```
define effort low | mid | high | time
```

For interactive runs it is recommended to run a prove with low effort first to get most of the easy results quickly, and then follow with a significantly higher value, like 1 hour, to obtain results for more difficult assertions, e.g.

```
define effort low

prove

define effort 1h

prove
```

The advantage of this strategy is that the user can already debug the quick failures while the tool spends time on the more difficult ones.

Increasing the effort is usually the first reaction on explored results.

Engines

IEV comes with a broad set of engines that are based on BDD, SAT or combinations of both. Currently we provide 14 different engines:

1. Axe

   Axe1 uses a BDD based forward reachability. It is suitable for small control-oriented designs. This engine can also be used in case the design under test has counters that have a deep state space

2. Axe2

   Axe2 is a variant of Axe that uses a BDD based forward-backward reachability algorithm

3. Bow

Bow1 is a SAT based engine and is capable of generating only Fail status for assertions and Pass status for covers. This engine uses an aggressive optimization method to reduce the complexity of large designs. This is the best engine for bug hunting.

4. Bow2

Bow2 is a variant of Bow that additionally restricts design behavior by applying pin constraints on input bit vectors. It does not report depth value for assertions with Explored status. This engine is suitable for bug hunting in designs with wide data paths.

5. Bow3

Bow3 is a variant of Bow that can perform better on smaller designs

6. Dagger

Dagger is a SAT based induction engine. It is a versatile engine and is suitable for a variety of design styles and sizes.

7. Hammer

Hammer is an abstraction refinement based engine and uses a combination of both SAT and BDD based techniques for model checking.

8. Saber

Saber is a newer SAT based engine that uses property driven reachability technique for the assertion verification. It is a versatile engine and is suitable for variety of design styles and sizes.

9. Spear

Spear uses a mixture of ATPG and BDD based algorithms for model checking.

10. Spear2

Spear2 uses a pure ATPG based algorighm for model checking.

11. Sword

Sword1 is a SAT based engine. It is a versatile engine and is suitable for avariety of design styles and sizes.

12. Sword2

Sword2 is a variation of Sword. This engine uses a different technique to handle constraints and is suitable for designs having simple constraints.

13. Sword3

Sword3 is a variant of sword that uses an aggressive optimization method to reduce design complexity designs. It is the most versatile engine and is suitable for a variety of design styles and sizes.

14. Sword4

Sword4 is a variant of Sword that uses the internal equivalence of design nodes to reduce the verification complexity. It is particularly useful for sequential equivalence checking.

All of the engines above support safety and liveness assertions.

In order to select a particular engine, type

```
define engine <engine>
```

If nothing is specified we use an auto selection of the 2 best proving and falsification engines: Sword 3 and Bow (subject to change).

Adding more engines to the proof is usually the second reaction on explored results

Halo

IEV provides automatic localization abstractions. They determine the radius of the subset of logic, which is tried for proving the check or falsifying a cover. If the trial is unsuccessful, the next bigger subset is chosen and so forth. This strategy often allows coming up with an assertion pass on a subset of the original cone of influence already. An assertion failure always requires to run this iteration up to the primary inputs, so we can create a legal counterexample.

There are 3 styles of Halo to choose from:

1. Sequential

   The radius is determined by back tracing to the next level of state elements. This is recommended if there is a high likelihood of an assertion to pass.

2. Hierarchical

   The radius is determined by back tracing to the next module boundary.

3. Off

   The first radius is including the full COI already. This works particularly well with engine bow when you expect a property to fail or a cover to pass.

In order to select a Halo, type

```
define halo seq | hier | off
```

Adding halos is usually the third reaction to explored results.

Engine Distribution

14 different engines, 3 halos – which are the optimal combination for my property? This is actually a question that we cannot easily answer. As a matter of fact, the best engine for a particular problem is as unpredictable as is the result of the assertion itself.

The solution to this dilemma is a clever parallelization strategy. IEV distribute the engines and halos on multiple CPUs in parallel, and whatever engine produces the first conclusive results aborts the other runs. The order of engines selected hereby is determined by and updated with the latest success rate of the particular engines in our performance test suite.

To turn on engine distribution, type

```
define engine auto_dist
```

This will incur up to 19 different engine halo combinations in version 12.1, depending on the number of CPUs available. Use this if you require running the ultimately best engine on a property. You can manually limit the combinations using

```
define auto_dist_max <number>
```

In practice a limit of 4 is a reasonable compromise for bigger testcases.

Vacuity

Vacuity check is run before any assertion check and make sure that the constraints do not conflict with each other, which could lead to invalid results. Because these check takes time, it is recommended to disable it once you are sure that your constraints will not change. You can disable vacuity check by

```
define vacuity off
```

Witness

IE automatically creates additional trigger and trace checks for each property to enable coverage and reachability analysis (e.g. to detect vacuous passes). These checks add to the runtime and can be disabled for overall runtime reduction using the following command:

```
define witness_check off
```

Property Distribution

On top of Engine distribution IEV provides Property based distribution using a standard load sharing infrastructure (LSF) and Sun Grid Engine (SGE). This means we distribute all engine/halo combinations of multiple properties over the assigned CPUs in parallel.

Set the multiple property distribution mode, using the following command:

```
define parallel_mode lsf | sge| local
```

Use the following command to define the number of properties, jobs, or licenses to run in parallel:

```
define parallel_max_limit <number> [unit]| auto (default)
```

Where number corresponds to an integer greater than 1 and unit corresponds to properties, jobs, or licenses.

The IEV GUI provides windows and wizards that help you setting up the LSF/SGE strings and let you monitor the individual engine/halo/property jobs in the system.

Utilizing this capability has the biggest measurable impact on overall runtime.

Clock Optimization

Clock optimization is a feature that improves performance significantly by eliminating activities on inactive clock edges of the design from the model to be verified by the engine. This is desired when you verify designs which operate synchronously on one edge of the clock.

You can check for success of this optimization using the following command:

```
report –optimization
```

If IEV failed to optimize clocks, it provides additional information why, so users can analyze and react to enable it. There is a dedicated whitepaper published on this topic. It is generally recommended to enforce clock optimization in standard invocations using the following setting, unless your intention is to perform asynchronous formal analysis:

```
define stop_verification clock_not_opt
```

Word Level Reduction

This feature automatically reduces the width of data path in case of data independent systems. This helps to minimize the complexity of the design. Turn it on using the following command.

```
define word_level_reduction on
```

The user guide mentions a couple limitations.

Clubbing

Clubbing is an alternative to property distribution and takes all the currently enabled properties and provides it as a clubbed problem to the engines. This is an expert level feature and can improve or

worsen the performance depending upon the design and assertions. It can make sense to improve runtime if the properties are relatively easy and share a lot of COI.

You can enable clubbing of assertions by issuing the following command:

```
define clubbing on
```

Constraint Minimization

Usually only a subset of constraints are actually required to obtain a valid result. With constraint minimization the tool reduces the set of constraints required to prove an assertion, which often improve the performance. It offers 2 different levels:

1. Static minimization

   The tool uses static analysis of the COI and consequently the constraints that have only clock or reset nets common with the existing COI are not selected.

2. Iterative minimization

   The tool uses a patented iterative algorithm leveraging ADS and Formal that additionally tries to remove constraints in a trial/error fashion. If it encounters a invalid result in these trials, it will restore the constraint, otherwise it drops it. That can ideally lead to the minimal required set of constraints required for a proof.

You can enable constraint minimization at prove, using the following command:

```
define minimize_constraints static | iterative
```

To view the minimized constraints, you can use the following command:

```
assert -show –dependent
```

it is recommended to apply iterative constraint minimization on explored assertions after all other features above have been explored.

Abstractions

There are currently several new automatic abstractions for the under development, which include memory abstractions, FIFO abstractions as well as counter abstractions. They are discussed in separate documents.