SIEMENS

Home > Verification Methodology Cookbooks > UVM

UVM Performance Guidelines

Although the UVM improves verification productivity, there are certain aspects of the methodology that should be used with caution, or perhaps not at all, when it comes to performance and scalability considerations.

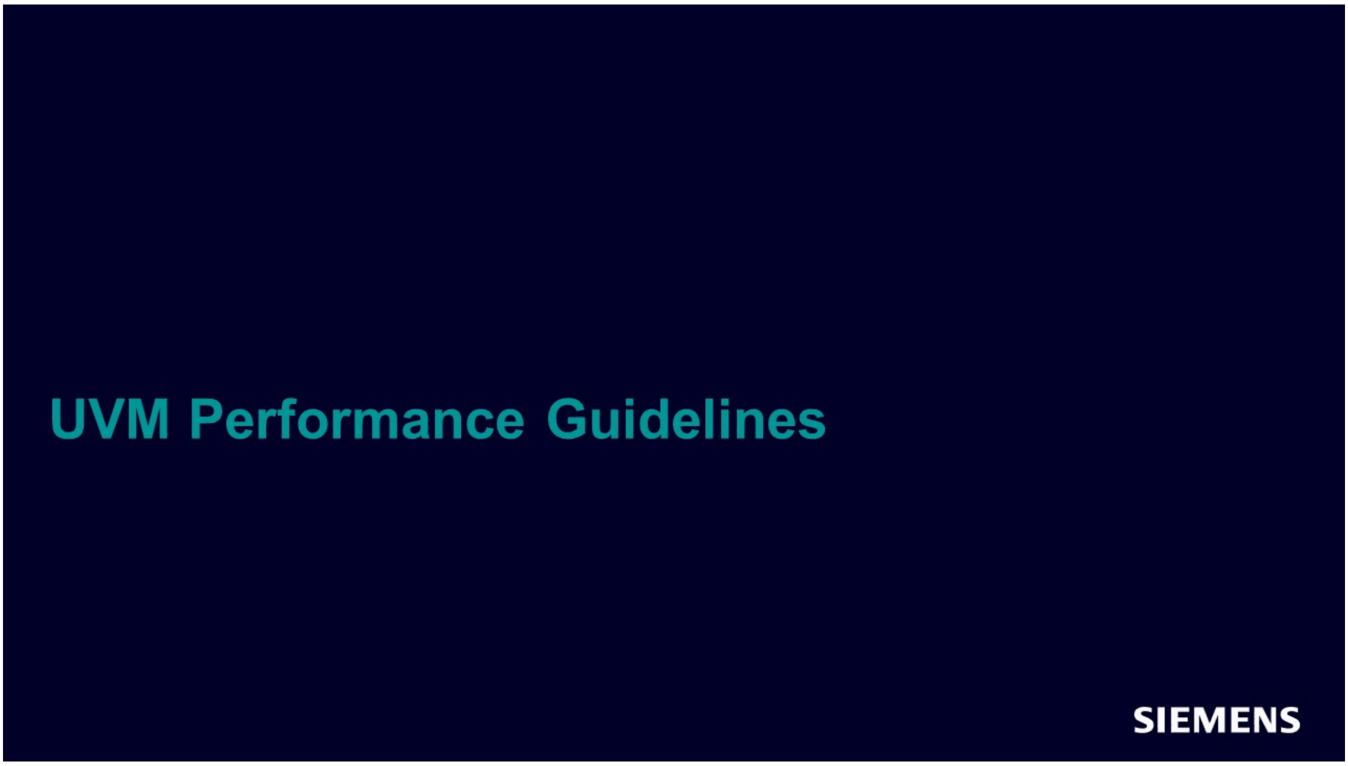
UVM - Universal Verification Methodology

Verification Methodology Team

Last Updated Mar 2014

UVM, SystemVerilog, Appendix, Guidelines, Performance, Beginner

Mark as viewed



During the simulation run time of a UVM testbench there are two distinct periods of activity. The first is the set of UVM phases that have to do with configuring, building and connecting up the testbench component hierarchy, the second is the run-time activity where all the stimulus and analysis activity takes place. The performance considerations for both periods of activity are separate.

These performance guidelines should be read in conjunction with the other methodology cookbook guidelines, there are cases where judgement is required to trade-off performance, re-use and scalability concerns.

UVM Testbench Configuration and Build Performance Guidelines

The general requirement for the UVM testbench configuration and build process is that it should be quick so that the run time phases can get going. With small testbenches containing only a few components - e.g. up to 10, the build process should be short, however when the testbench grows in size beyond this then the overhead of using certain features of the UVM start to become apparent.

With larger testbenches with 100s, or possibly 1000s of components, the build phase will be noticeably slower. The guidelines in this section apply to the full spectrum of UVM testbench size, but are most likely to give most return as the number of components increases.

Avoid auto-configuration

Auto-configuration is a methodology inherited from the OVM where a component's configuration variables are automatically set to their correct values from variables that have been set up using set_config_int(), set_config_string(), uvm_config_db #(..)::set() etc at a higher level of the component hierarchy.

In order to use auto-configuration, field macros are used within a component and the super.build_phase() method needs to be called during the build_phase(), the auto-configuration process then attempts to match the fields in the component with entries in the configuration database via a method in uvm_component called apply_config_settings(). From the performance point of view, this is VERY expensive and does not scale.

Lower Performance Version

```
class my_env extends uvm_component;

bit has_axi_agent;
bit has_ahb_agent;
string system_name;

axi_agent m_axi_agent;
ahb_agent m_ahb_agent;

// Required for auto-configuration
`uvm_component_utils_begin(my_env)
  `uvm_field_int(has_axi_agent, UVM_DEFAULT)
  `uvm_field_int(has_ahb_agent, UVM_DEFAULT)
  `uvm_field_string(system_name, UVM_DEFAULT)
`uvm_component_utils_end

function new(string name = "my_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
  super.build_phase(phase); // Auto-configuration called here
  if(has_axi_agent == 1) begin
    m_axi_agent = axi_agent::type_id::create("m_axi_agent", this);
  end
  if(has_ahb_agent == 1) begin
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
  `uvm_info("build_phase", $sformatf("%s built", system_name))
endfunction: build_phase

endclass: my_env
```

Higher Performance Version

```
class my_env extends uvm_component;

my_env_config cfg;

axi_agent m_axi_agent;
ahb_agent m_ahb_agent;

`uvm_component_utils(my_env)

function new(string name = "my_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
  // Get the configuration, note class variables not required
  if(!uvm_config_db #(my_env_config)::get(this, "", "my_env_config", cfg)) begin
    `uvm_error("build_phase", "Unable to find my_env_config in uvm_config_db")
  end
  if(cfg.has_axi_agent == 1) begin
    m_axi_agent = axi_agent::type_id::create("m_axi_agent", this);
  end
  if(cfg.has_ahb_agent == 1) begin
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
```

```
    `uvm_info("build_phase", $sformatf("%s built", cfg.system_name))
  endfunction: build_phase

  endclass: my_env
```

The recommended practice is not to use field macros in a component, and to not call super.build_phase() if the class you are extending is from a UVM component base class such as uvm_component. Even then, when a component does not have a build_phase() method implementation, the default build_phase() from the uvm_component base class will be called which will attempt to do auto-configuration.

In UVM 1.1b, a fix was added that stops the apply_config_settings() method from continuing if there are no field macros in the component, this speeds up component build, but it is more efficient to avoid this method being called altogether.

Minimize the use of the uvm_config_db

The uvm_config_db is a database, as with any database it takes longer to search as it grows in size. The uvm_config_db is based on the uvm_resource and the uvm_resource_db classes. The uvm_resource_db uses regular expressions and the component hierarchy strings to make matches, it attempts to check every possible match and then returns the one that is closest to the search, this is expensive and the search time increases exponentially as the database grows.

Therefore, the uvm_config_db should be used sparingly, if at all. This also applies to the set/get_config_xxx() methods since they in turn are based on the uvm_config_db.

Use configuration objects to pass configuration data to components

One way to minimize the number of uvm_config_db entries is to group component configuration variables into a configuration object. That way only one object needs to be set() in the uvm_config_db. This has reuse benefits and is the recommended way to configure reusable verification components such as agents.

Lower Performance Version

```
class static_test extends uvm_test;

// Test that builds an env containing an AXI agent

virtual axi_if AXI; // Used by the AXI agent
```

```
// Only consider the build method:
function void build_phase(uvm_phase phase);
  // Configuration code for the AXI agent
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI", AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  uvm_config_db #(virtual axi_if)::set(this, "env.axi_agent*", "v_if", AXI);
  uvm_config_db #(uvm_active_passive_enum)::set(
                    this, "env.axi_agent*", "is_active", UVM_ACTIVE);
  uvm_config_db #(int)::set(this, "env.axi_agent*", "max_burst_size", 16);
  // Other code

endfunction: build_phase

endclass: static_test

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration parameters:
virtual axi_if AXI;
uvm_active_passive_enum is_active;
int max_burst_size;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI", AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  if(!uvm_config_db #(uvm_active_passive_enum)::get(this,
                                    "", "is_active", is_active)) begin
    `uvm_error("build_phase", "is_active not found in uvm_config_db")
  end
  if(!uvm_config_db #(int)::get(
                    this, "", "max_burst_size", max_burst_size)) begin
    `uvm_error("build_phase", "max_burst_size not found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

function void connect_phase(uvm_phase phase);
  monitor.AXI = AXI;
  if(is_active == UVM_ACTIVE) begin
    driver.AXI = AXI;
    driver.max_burst_size = max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

Higher Performance Version

```
// Additional agent configuration class:
class axi_agent_config extends uvm_object;

`uvm_object_utils(axi_agent_config)

virtual axi_if AXI;
uvm_active_passive_enum is_active = UVM_ACTIVE;
int max_burst_size = 64;

function new(string name = "axi_agent_config");
  super.new(name);
endfunction

endclass: axi_agent_config

class static_test extends uvm_test;

// Test that builds an env containing an AXI agent

axi_agent_config axi_cfg; // Used by the AXI agent

// Only consider the build method:
function void build_phase(uvm_phase phase);
  // Configuration code for the AXI agent
  axi_cfg = axi_agent_config::type_id::create("axi_cfg");
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI",
                                          axi_cfg.AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  axi_cfg.is_active = UVM_ACTIVE;
  axi_cfg.max_burst_size = 16;
  uvm_config_db #(axi_agent_config)::set(this,
                     "env.axi_agent*", "axi_agent_config", axi_cfg);
  // Other code

endfunction: build_phase

endclass: static_test

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                  "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object not
                                          found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
```

```
endfunction: build_phase

function void connect_phase(uvm_phase phase);
  monitor.AXI = cfg.AXI;
  if(cfg.is_active == UVM_ACTIVE) begin
    driver.AXI = cfg.AXI;
    driver.max_burst_size = cfg.max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

The higher performance version of the example uses one uvm_config_db #(...)::set() call and two get() calls, compared with three set() and four get() calls in the lower performance version. There are also just two uvm_config_db entries compared to four. With a large number of components, this form of optimization can lead to a considerable performance boost.

Minimize the number of uvm_config_db #(...)::get() calls

The process of doing a get() from the uvm_config_db is expensive, and should only be used when really necessary. For instance, in an agent, it is only really necessary to get() the configuration object at the agent level and to assign handles to the sub-components from there. It is an uneccessary overhead to have separate get() calls inside the driver and monitor components.

Lower Performance Version

```
// Agent configuration class - configured and set() by the test class
class axi_agent_config extends uvm_object;

`uvm_object_utils(axi_agent_config)

virtual axi_if AXI;
uvm_active_passive_enum is_active = UVM_ACTIVE;
int max_burst_size = 64;

function new(string name = "axi_agent_config");
  super.new(name);
endfunction

endclass: axi_agent_config

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;
```

```
function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                    "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                           not found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

endclass: axi_agent

// The axi monitor:
class axi_monitor extends uvm_component;

axi_if AXI;
axi_agent_config cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                     "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                             not found in uvm_config_db")
  end
  AXI = cfg.AXI;
endfunction: build_phase

endclass: axi_monitor

// The axi driver:
class axi_monitor extends uvm_component;

axi_if AXI;
int max_burst_size;
axi_agent_config cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                       "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object
                                              not found in uvm_config_db")
  end
  AXI = cfg.AXI;
  max_burst_size = cfg.max_burst_size;
endfunction: build_phase

endclass: axi_driver
```

Higher Performance Version

```
// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;
```

```
  axi_driver driver;
  axi_sequencer sequencer;
  axi_monitor monitor;

  function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(axi_agent_config)::get(this,
                                        "", "axi_agent_config", cfg)) begin
      `uvm_error("build_phase", "AXI agent config object
                                               not found in uvm_config_db")
    end
    monitor = axi_monitor::type_id::create("monitor", this);
    if(cfg.is_active == UVM_ACTIVE) begin
      driver = axi_driver::type_id::create("driver", this);
      sequencer = axi_sequencer::type_id::create("sequencer", this);
    end
  endfunction: build_phase

  // Direct assignment to the monitor and driver variables
  // from the configuration object variables.
  function void connect_phase(uvm_phase phase);
    monitor.AXI = cfg.AXI;
    if(cfg.is_active == UVM_ACTIVE) begin
      driver.AXI = cfg.AXI;
      driver.max_burst_size = cfg.max_burst_size;
    end
  endfunction: connect_phase

  endclass: axi_agent

  // The axi_monitor and axi_driver are implemented without
  // a uvm_config_db #()::get()
```

The higher performance version has two fewer calls to the uvm_config_db #()::get() method, which when multiplied by a large number of components can lead to a performance improvement.

Use specific strings with the uvm_config_db set() and get() calls

The regular expression algorithm used in the search attempts to get the closest match based on the UVM component's position in the testbench hierarchy and the value of the key string. If wildcards are used in either the set() or get() process, then this adds ambiguity to the search and makes it more expensive.

For instance, setting the context string to "*" means that the entire component hierarchy will be searched for uvm_config_db settings before a result is returned.

Lower Performance Version

```
// In one component, setting config for env.sb
sb_cfg = sb_config::type_id::create("sb_cfg");
// Configure content of sb_cfg ...
umv_config_db #(sb_config)::set(this, "*", "*_config", sb_cfg);

// In the env.sb component:
sb_config cfg;
if(!uvm_config_db #(sb_config)::get(this, "", "*_config", cfg)) begin
  `uvm_error(...)
end
```

Higher Performance Version

```
// In one component, setting config for env.sb
sb_cfg = sb_config::type_id::create("sb_cfg");
// Configure content of sb_cfg ...
umv_config_db #(sb_config)::set(this, "env.sb", "sb_config", sb_cfg);

// In the env.sb component:
sb_config cfg;
if(!uvm_config_db #(sb_config)::get(this, "", "sb_config", cfg)) begin
  `uvm_error(...)
end
```

In the higher performance version of this code, the scope is very specific and will only match on the single component for a single key, this cuts downs the search time in the uvm_config_db

Minimize the number of virtual interface handles passed via uvm_config_db from the TB module to the UVM environment

Consolidate the virtual interface handles into a single configuration object

An alternative to using a package to pass the virtual interface handles from the testbench top level module to the UVM test, is to create a single configuration object that contains all the virtual interface handles and to make the virtual interface assignments in the top level module before setting the configuration object in the uvm_config_db. This reduces the number of uvm_config_db entries used for passing virtual interface handles down to one.

```
// Virtual interface configuration object:
class vif_handles extends uvm_object;
`uvm_object_utils(vif_handles)

virtual axi_if AXI;
virtual ddr2_if DDR2;

endclass: vif_handles

// In the top level testbench module:
module top_tb;
```

```
import uvm_pkg::*;
import test_pkg::*;

// Instantiate the static interfaces:
axi_if AXI();
ddr2_if DDR2();

// Virtual interface handle container object:
vif_handles v_h;

// Hook up to DUT ....

// UVM initial block:
initial begin
  // Create virtual interface handle container:
  v_h = vif_handles::type_id::create("v_h");
  // Assign handles
  v_h.AXI = AXI;
  v_h.DDR2 = DDR2;
  // Set in uvm_config_db:
  uvm_config_db #(vif_handles)::set("uvm_test_top", "", "V_H", vh);
  run_test();
end

endmodule: top_tb
```

Pass configuration information through class hierarchical references

The ultimate in minimizing the use of the uvm_config_db is not to use it at all. It is perfectly possible to pass handles to configuration objects through the class hierarchy at build time. Using handle assignments is the most efficient way to do this.

Lower Performance Version

```
// In the test, with the env configuration object containing
// nested configuration objects for its agents:
function void build_phase(uvm_phase phase);
  env_cfg = env_config_object::type_id::create("env_cfg");
  // Populate the env_cfg object with axi_cfg, ddr2_cfg etc
  env = test_env::type_id::create("env");
  uvm_config_db #(env_config_object)::set(this, "env", "env_cfg", env_cfg);
  // ...
endfunction: build_phase

// In the env, building the axi and ddr2 agents:
env_config_object cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(env_config_object)::get(this,
                                               "", "env_cfg", cfg)) begin
    `uvm_error(...)
  end
  // Create AXI agent and set configuration for it:
  axi = axi_agent::type_id::create("axi", this);
```

```
    uvm_config_db #(axi_agent_config)::set(this,
                                  "axi", "axi_agent_config", cfg.axi_cfg);
    // Also for the DDR2 agent:
    ddr2 = ddr2_agent::type_id::create("ddr2", this);
    uvm_config_db #(ddr2_agent_config)::set(this,
                               "ddr2", "ddr2_agent_config", cfg.ddr2_cfg);
    // etc
  endfunction: build_phase
```

Higher Performance Version

```
// In the test, with the env configuration object containing
// nested configuration objects for its agents:
function void build_phase(uvm_phase phase);
  env_cfg = env_config_object::type_id::create("env_cfg");
  // Populate the env_cfg object with axi_cfg, ddr2_cfg etc
  env = test_env::type_id::create("env");
  // Assign the env configuration object handle directly:
  env.cfg = env_cfg;
  // ...
endfunction: build_phase

// In the env, building the axi and ddr2 agents:
env_config_object cfg;

function void build_phase(uvm_phase phase);
  // Create AXI agent and set configuration for it:
  axi = axi_agent::type_id::create("axi", this);
  // Assign axi agents configuration handle directly:
  axi.cfg = cfg.axi_cfg;
  // Also for the DDR2 agent:
  ddr2 = ddr2_agent::type_id::create("ddr2", this);
  ddr2.cfg = cfg.ddr2_cfg;
  // etc
endfunction: build_phase
```

The higher performance example avoids the use of the uvm_config_db altogether, providing the ultimate configuration and build performance enhancement. The impact of using this approach is that it requires the assignments to be chained together; that it requires the agent code to test for a null config object handle before attempting to get the configuration object handle; and that any stimulus hierarchy needs to take care of getting handles to testbench resources such as register models.

Another major consideration with this direct approach to the assignment of configuration object handles is that if VIP is being re-used, it may well be implemented with the expectation that its configuration object will be set in the uvm_config_db. This means that there may have to be some use of the uvm_config_db to support the reuse of existing VIP.

Minimize the use of the UVM Factory

The UVM factory is there to allow UVM components or objects to be overriden with derived objects. This is a powerful technique, but whenever a component is built a lookup has to be made in a table to determine which object type to construct. If there are overrides, this lookup becomes more complicated and there is a performance penalty. Try to manage the factory overrides used to reduce the overhead of the lookup.

UVM Testbench Run-Time Performance Guidelines

The guidelines presented here represent areas of the UVM which have been seen to cause performance issues during the run-time phases of the testbench, they are mostly concerned with stimulus generation. There are other SystemVerilog coding practices that can also followed to enhance run-time performance and these are described in the SystemVerilog Performance Guidelines article.

Avoid polling the uvm_config_db for changes

Do not use the uvm_config_db to communicate between different parts of the testbench, for instance by setting a new variable value in one component and getting it inside a poll loop in another. It is far more efficient for the two components to have a handle to a common object and to reference the value of the variable within the object.

Lower Performance Version

```
// In a producer component setting the value inside a loop:
int current_id = 0;

forever begin
  // Lots of work making a transfer occur
  // Communicate the current id:
  uvm_config_db #(int)::set(null, "*", "current_id", current_id);
  current_id++;
end

// In a consumer component looking out for the current_id value
int current_id;

forever begin
   uvm_config_db #(int)::wait_modified(this, "*", "current_id");
   if(!uvm_config_db #(int)::get(this,
                              "", "current_id", current_id)) begin
     `uvm_error( ....)
   end
   // Lots of work to track down a transaction with the current_id
end
```

Higher Performance Version

```
// Config object containing current_id field:
packet_info_cfg pkt_info =
                        packet_info_cfg::type_id::create("pkt_info");
// This created in the producer component and the consumer component
// has a handle to the object:

// In the producer component:
forever begin
  // The work resulting in a current_id update
  pkt_info.current_id = current_id;
  current_id++;
end

// In the consumer component:
forever begin
  @(pkt_info.current_id);
  // Start working with the new id
end
```

The principle at work in the higher performance version is that the current_id information is inside an object. Both the consumer and the producer components share the handle to the same object, therefore when the producer object makes a change to the current_id field, it is visible to the consumer component via the handle. This avoids the use of repeated set() and get() calls in the uvm_config_db and also the use of the expensive wait_modified() method.

Do not use the UVM field macros in transactions

The UVM field macros may seem like a convenient way to ensure that the various do_copy(), do_compare() methods get implemented, but this comes at a heavy cost in terms of performance. This becomes very evident if your testbench starts to use sequence_items heavily.

Lower Performance Version

```
// APB Bus sequence_item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we;

// Field macros:
`uvm_object_utils_begin(apb_seq_item)
  `uvm_field_int(addr, UVM_DEFAULT)
  `uvm_field_int(data, UVM_DEFAULT)
  `uvm_field_enum(we, apb_opcode_e, UVM_DEFAULT)
`uvm_object_utils_end

function new(string name = "apb_seq_item");
  super.new(name);
endfunction
```

```
endclass: apb_seq_item
```

Higher Performance Version

```
// APB Bus sequence_item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we;

`uvm_object_utils(apb_seq_item)

function new(string name = "apb_seq_item");
  super.new(name);
endfunction

// Sequence Item convenience method prototypes:
extern function void do_copy(uvm_object rhs);
extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);
extern function string convert2string();
extern function void do_print(uvm_printer printer);
extern function void do_record(uvm_recorder recorder);
extern function void do_pack();
extern function void do_unpack();
endclass: apb_seq_item
```

Although the lower performance code example looks more compact, compiling with an -epretty flag will reveal that they expand out into many lines of code. The higher performance example shows the templates for the various uvm_object convenience methods which should be implemented manually, this will always improve performance and enhance debug should you need it.

The definitive guide on the trade-offs involved in using or not using these and the various other UVM macros can be found here.

Minimize factory overrides for stimulus objects

The UVM factory can be used to override or change the type of object that gets created when a object handle's ::type_id::create() method is called. During stimulus generation this could be applied to change the behavior of a sequence or a sequence_item without rewriting the testbench code. However, this override capability comes at a cost in terms of an extended lookup in the factory each time the object is created. To reduce the impact of creating an object overridden in the factory, create the object once and then clone it each time it is used to avoid using the factory.

Lower Performance Version

```
// apb_seq_item has been factory overridden with apb_seq_error_item
class reg_bash_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item item;

  repeat(200) begin
    item = apb_seq_item::type_id::create("item");
    start_item(item);
    assert(item.randomize() with {addr inside {[`reg_low:`reg_high]};});
    finish_item(item);
endtask:body

endclass: reg_bash_seq
```

Higher Performance Version

```
// apb_seq_item has been factory overridden with apb_seq_error_item
class reg_bash_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item original_item = apb_seq_item::type_id::create("item");
  apb_seq_item item;

  repeat(200) begin
    $cast(item, original_item.clone());
    start_item(item);
    assert(item.randomize() with {addr inside {[`reg_low:`reg_high]};});
    finish_item(item);
endtask:body

endclass: reg_bash_seq
```

The higher performance example only makes one factory create call, and uses clone() to create further copies of it, so saving the extended factory look-up each time that is expended each time round the generation loop in the lower performance example.

Avoid embedding covergroups in transactions

Embedding a covergroup in a transaction adds to its memory footprint, it also does not make sense since the transaction is disposable. The correct place to collect coverage is in a component. Transactional coverage can be collected by sampling a covergroup in a component based on transaction content.

Lower Performance Version

```
// APB Sequence item
class apb_seq_item extends uvm_sequence_item;
```

```
bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we

covergroup register_space_access_cg;

ADDR_RANGE: coverpoint addr[7:0];
OPCODE: coverpoint we {
  bins rd = {APB_READ};
  bins_wr = {APB_WRITE};
}
ACCESS: cross ADDR_RANGE, OPCODE;

endgroup: register_space_access_cg;

function void sample();
  register_space_access_cg.sample();
endfunction: sample

// Rest of the sequence item ...

endclass: apb_seq_item

// Sequence producing the sequence item:
class bus_access_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item apb_item = apb_seq_item::type_id::create("apb_item");

  repeat(200) begin
    start_item(apb_item);
    assert(apb_item.randomize());
    apb_item.sample();
    finish_item(apb_item);
  end
endtask: body
```

Higher Performance Version

```
// apb_seq_item implemented without a covergroup
// Therefore bus_access_seq does not sample covergroup
// Sampling coverage in the driver:
class apb_coverage_driver extends apb_driver;

covergroup register_space_access_cg() with
                        function sample(bit[7:0] addr, apb_opcode_e we);

ADDR_RANGE: coverpoint addr;
OPCODE: coverpoint we {
  bins rd = {APB_READ};
  bins_wr = {APB_WRITE};
}
ACCESS: cross ADDR_RANGE, OPCODE;

endgroup: register_space_access_cg;

task run_phase(uvm_phase phase);
  apb_seq_item apb_item;
```

```
    forever begin
      seq_item_port.get(apb_item);
      register_space_access_cg.sample(apb_item.addr[7:0], apb_item.we);
      // Do the signal level APB cycle
      seq_item_port.item_done();
    end

  endtask: run_phase

  // ....
  endclass: apb_coverage_driver
```

The lower performance example shows the use of a covergroup within a transaction to collect input stimulus functional coverage information. This adds a memory overhead to the transaction that is avoided by the higher performance example which collects coverage in a static component based on the content of the transaction.

Use the UVM reporting macros

The raw UVM reporting methods do not check the verbosity of the message until all of the expensive string formatting operations in the message assembly have completed. The `uvm_info(), `uvm_warning(), `uvm_error(), and `uvm_fatal() macros check the message verbosity first and then only do the string formatting if the message is to be printed.

Lower Performance Version

```
  function void report_phase(uvm_phase phase);
  if(errors != 0) begin
    uvm_report_error("report_phase", $sformatf(
                   %0d errors found in %0d transfers", errors, n_tfrs));
  end
  else if(warnings != 0) begin
    uvm_report_warning("report_phase", $sformatf(
             "%0d warnings issued for %0d transfers", warnings, n_tfrs));
  end
  else begin
    uvm_report_info("report_phase", $sformatf(
                               "%0d transfers with no errors", n_tfrs));
  end
  endfunction: report_phase
```

Higher Performance Version

```
  function void report_phase(uvm_phase phase);
  if(errors != 0) begin
    `uvm_error("report_phase", $sformatf(
                    "%0d errors found in %0d transfers", errors, n_tfrs))
  end
```

```
else if(warnings != 0) begin
  `uvm_warning("report_phase", $sformatf(
             "%0d warnings issued for %0d transfers", warnings, n_tfrs))
end
else begin
  `uvm_info("report_phase", $sformatf(
                     "%0d transfers with no errors", n_tfrs), UVM_MEDIUM)
end
endfunction: report_phase
```

In the example shown, the same reports would be generated in each case, but if the verbosity settings are set to suppress the message, the higher performance version would check the verbosity before generating the strings. In a testbench where there are many potential messages and the reporting verbosity has been set to low, this can have a big impact on performance, especially if the reporting occurs frequently.

Do not use the uvm_printer class

The uvm_printer is a convenience class, originally designed to go with the use of field macros in order to print out component hierarchy or transaction content in one of several formats. The class comes with a performance overhead and its use can be avoided by using the convert2string() method for objects. The convert2string() method returns a string that can be displayed or printed using the UVM messaging macros.

Lower Performance Version

```
apb_seq_item bus_req = abp_seq_item::type_id::create("bus_req");

repeat(20) begin
  start_item(bus_req);
  assert(bus_req.randomize());
  finish_item(bus_req);
  bus_req.print();
end
```

Higher Performance Version

```
apb_seq_item bus_req = abp_seq_item::type_id::create("bus_req");

repeat(20) begin
  start_item(bus_req);
  assert(bus_req.randomize());
  finish_item(bus_req);
  `uvm_info("BUS_SEQ", bus_req.convert2string(), UVM_HIGH)
end
```

Note also that the print() method calls $display() without checking verbosity settings.

Avoid the use of get_xxx_by_name() in UVM register code

Using the get_field_by_name(), or the get_register_by_name() functions involves a regular expression search of all of the register field name or register name strings in the register model to return a handle to a field or a register. As the register model grows, this search will become more and more expensive.

Use the hierarchical path within the register model to access register content, it is far more efficient as well as being a good way to make register based stimulus reusable.

Lower Performance Version

```
task set_txen_field(bit[1:0] value);
  uvm_reg_field txen;

  txen = rm.control.get_field_by_name("TXEN");
  txen.set(value);
  rm.control.update();
endtask: set_txen_field
```

Higher Performance Version

```
task set_txen_field(bit[1:0] value);
  rm.control.txen.set(value);
  rm.control.update();
endtask: set_txen_field
```

The higher performance version of the set_txen_field avoids the expensive regular expression lookup of the field's name string.

Minimize the use of get_registers() or get_fields() in UVM register code

These calls, and others like them return return queues of object handles, this is for convenience since a queue is an unsized array. Calling these methods requires the queue to be populated which can be an overhead if the register model is a reasonable size. Repeated calls of these methods is pointless, they should only need to be called once or twice within a scope.

Lower Performance Version

```
uvm_reg regs[$];
randc int idx;
int no_regs;
```

```
repeat(200) begin
  regs = rm.encoder.get_registers();
  no_regs = regs.size();
  repeat(no_regs) begin
    tassert(this.randomize() with {idx =< no_regs;});
    assert(regs[idx].randomize());
    regs[idx].update();
  end
end
```

Higher Performance Version

```
uvm_reg regs[$];
randc int idx;
int no_regs;

regs = rm.encoder.get_registers();
repeat(200) begin
  regs.shuffle();
  foreach(regs[i]) begin
    assert(regs[i].randomize());
    regs[i].update();
  end
end
```

The higher performance version of the code only does one get_registers() call and avoids the overhead associated with the repeated call in the lower performance version.

Use UVM objections, but wisely

The purpose of raising a UVM objection is to prevent a phase from completing until a thread is ready for it to complete. Raising and dropping objections causes the component hierarchy to be traversed, with the objection being raised or dropped in all the components all the way to the top of the hierarchy. Therefore, raising and lowering an objection is expensive, becoming more expensive as the depth of the testbench hierarchy increases.

Objections should only be used by controlling threads, and the proper place to put objections is either in the run-time method of the top level test class, or in the body method of a virtual sequence. Using them in any other place is likely to be unecessary and also cause a degradation in performance.

Lower Performance Version

```
// Sequence to be called:
class adpcm_seq extends uvm_sequence #(adpcm_seq_item);
//...
```

```
task body;
  uvm_objection objection = new("objection");
  adpcm_seq_item item = adpcm_seq_item::type_id::create("item");

  repeat(10) begin
    start_item(item);
    assert(item.randomize());
    objection.raise_objection(this);
    finish_item(item);
    objection.drop_objection(this);
  end

// Inside the virtual sequence
adpcm_sequencer ADPCM;

task body;
  adpcm_seq do_adpcm = adpcm_seq::type_id::create("do_adpcm");
  do_adpcm.start(ADPCM);
endtask
```

Higher Performance Version

```
// Sequence to be called:
class adpcm_seq extends uvm_sequence #(adpcm_seq_item);
//...
task body;
  adpcm_seq_item item = adpcm_seq_item::type_id::create("item");

  repeat(10) begin
    start_item(item);
    assert(item.randomize());
    finish_item(item);
  end

// Inside the virtual sequence
adpcm_sequencer ADPCM;

task body;
  uvm_objection objection = new("objection");
  adpcm_seq do_adpcm = adpcm_seq::type_id::create("do_adpcm");
  objection.raise_objection(ADPCM);
  do_adpcm.start(ADPCM);
  objection.drop_objection(ADPCM);
endtask
```

In the higher performance version of the code, the objection is raised at the start of the sequence and dropped at the end, bracketing in time all the sequence_items sent to the driver, this is far more efficient than raising an objection per sequence_item.

Minimize the use of UVM call-backs

The implementation of call-backs in the UVM is expensive both in terms of the memory used and the code associated with registering and executing them. The complications arise mainly from the fact that the order in which the call-backs are registered is preserved. For performance, avoid the use of UVM call-backs by using alternative approaches to achieve the same functionality.

For example, register accesses can be recorded and viewed using transaction viewing either by extending the uvm_reg class or by using a call-back class.
The class extended from uvm_reg overloads the pre_read() and pre_write() methods to begin a transaction when a register read() or write() method is called, and overloads the post_read() and the post_write() methods to end the transaction when the register transfer has completed. This will result in a transaction being recorded for each register access, provided the extended class is used as the base class for the register model.

The alternative is to use a uvm_reg_cbs class which contains call-backs for the uvm_reg pre_read(), pre_write(), post_read() and post_write() methods. As with the extended class, the pre_xxx() methods start recording a transaction and the post_xxx() methods end recording transactions. A call back class object is then registered for each register using the package function enable_reg_recording().

Lower Performance Version Using Call Backs

```
//
// Call-Back class for recording register transactions:
//
class record_reg_cb extends uvm_reg_cbs;

  virtual task pre_write(uvm_reg_item rw);

  endtask

  virtual task post_write(uvm_reg_item rw);

  endtask

  virtual task pre_read(uvm_reg_item rw);

  endtask

  function void do_record(uvm_recorder recorder);

  endfunction

endclass : record_reg_cb

//
// Package function for enabling recording:
//
function void enable_reg_recording(uvm_reg_block reg_model,
                                   reg_recording_mode_t record_mode = BY_FIELD);
  uvm_reg regs[$];
```

```
    record_reg_cb reg_cb;

    //Set the recording mode
    uvm_config_db #(reg_recording_mode_t)::set(
                             null,"*","reg_recording_mode", record_mode);

    //Get the queue of registers
    reg_model.get_registers(regs);

    //Assign a callback object to each one
    foreach (regs[ii]) begin
      reg_cb = new({regs[ii].get_name(), "_cb"});
      uvm_reg_cb::add(regs[ii], reg_cb);
    end

    reg_cb = null;

    uvm_reg_cb::display();

  endfunction : enable_reg_recording
```

Higher Performance Version Using Class Extension

```
  //
  // Extension of uvm_reg enables transaction recording
  //
  class record_reg extends uvm_reg;

    virtual task pre_write(uvm_reg_item rw);

    endtask

    virtual task post_write(uvm_reg_item rw);

    endtask

    virtual task pre_read(uvm_reg_item rw);

    endtask

    function void do_record(uvm_recorder recorder);

    endfunction

  endclass : record_reg
```

The main argument for using call-backs in this case is that it does not require that the register model be used with the extended class, which means that it can be 'retro-fitted' to a register model that uses the standard UVM uvm_reg class. However, this comes at the cost of a significant overhead - there is an additional call-back object for each register in the register model and the calling of the transaction recording methods involves indirection through the UVM infrastructure to call the methods within the call-back object.

Given that a register model is likely to be generated, and that there could be thousands of registers in larger designs then using the extended record_reg class will deliver higher performance with minimum inconvenience to the user.

Previous Next

UVM Navigation

UVM Performance Guidelines

Recent Forum Discussions About UVM Performance Guidelines

Mask "Report counts by id" from test's UVM Report Summary UVM

Configuration UVM

If tool is Event based the how to make our uvm environment compatble ... U...

Pre_randomize usage in SystemVerilog performance guidelines UVM

UVM Accelerated TB (Co-Emulation) UVM

Ask a Question

Cloud

Design, Manufacturing and PLM Software

Electronic Design Automation

Insights Hub

Mendix

How to Buy

Buying with Siemens

Buy Online

Partners

Academics

Renewals

Siemens

About Us

Careers

Community

Events

Leadership

News and Press

Trust Center

Contact

VA - Contact Us

HLS - Contact Us

PLM - Contact Us

EDA - Contact Us

Worldwide Offices

Support Center

Provide Feedback

Report Piracy

Terms of Use Privacy Policy Cookie Statement DMCA Whistleblowing