Meld VM Architecture

Flavio Cruz

May 17, 2013

1 Directories

In this section I'll give a brief overview of every main directory in the virtual machine code. After this section, I'll give some details about further important concepts and a few "how to"'s for accomplishing certain tasks.

1.1 vm

This directory implements the core of the virtual machine. It is where the execution engine, the matching engine, byte code reading and tuple definitions are implemented.

exec This is where the instructions are executed.

external Every single external function must be registered in this class.

instr This module defines all the instructions and instruction sizes. Code for printing byte-code is also included.

predicate This class defines the predicate: its name, arguments, types, etc.

program The class `program` is responsible for reading a byte-code file and then instantiating all predicates and rules.

rule The class `rule` represents a rule. That includes the byte-code, predicates used, string representation, etc.

match Represents a matching for tuples that need to be retrieved from the database.

all This class encapsulates every information that is used during the execution of an instance of the virtual machine. Allows you to know how many threads are running, the contents of the database, the current program, etc.

rule_matcher A rule matcher contains information about currently active facts of some node. When a certain subset of facts are present, the rule matcher may activate rules to be executed.

state The `state` class implements rule matching and calls the `exec` module when it wants to run a rule.

types Utilities and functions to handle argument types of predicates.

tuple This represents a tuple in the system. A tuple is composed of a predicate plus an array of argument values. Argument values are defined in `defs.hpp` by the union `tuple_field`.

1.2 db

database Stores all nodes in the program.

tuple Implements a `simple_tuple` which is a tuple and a reference count. It is used in the database when storing tuples.

trie Implements a trie data structure that stores `simple_tuple`'s.

node The `node` class instantiates a trie for each predicate. It stores all the tuple information of a node.

tuple_aggregate Stores all `agg_configuration`'s of a predicate. If a predicate uses an aggregate, it will have a `tuple_aggregate` to store all its configurations. For example, predicate `f(node, int, min int)` will have as many configurations as the number of different integers in the second argument.

agg_configuration Stores tuples for a single aggregate configuration. In the previous example we could store facts `f(2, 3)`, `f(2, 4)` for the configuration `f(2, _)`.

1.3 sched

A *scheduler* represents a particular evaluation strategy for running the underlying graph model that represents the Meld program. For example, we may implement a scheduler that is able to run concurrently with other threads. Please remember that one scheduler is instantiated for each thread.

base The base scheduler from where every schedulers inherits from.

serial This is a special scheduler that is intended to be run with only thread. It performs sequential execution of programs.

serial_ui Very similar to the one above, however the scheduler also communicates with a webserver that is showing the results of execution.

sim This one may run several threads. It connects with the blinky blocks simulator to send new events and receive new facts about what happens to the simulated blinky block.

types The different types of schedulers. When adding a new one, we must add a new definition here.

thread Contains several utilities for threaded schedulers, namely termination barriers and thread state.

mpi Utilities for handling MPI-based schedulers. This is a somewhat abandoned directory.

1.4 process

After the `exec` module, this is the second most important component of the VM. Here, we setup the execution threads and other processes to run the VM.

machine A `machine` class is the main class that needs to be instantiated to run a particular meld program. It accepts as arguments the program file, number of threads and type of scheduler.

remote Represents a remote VM that is cooperating with the VM to run a program. This is related to MPI.

router Contains a set of `remote`'s that represent remote VMs. Also allows message sending and receiving to those remote.

1.5 mem

This is the memory subsystem of the Virtual Machine. It's responsible for allocating memory objects and avoid contention when using threads.

allocator Implements a memory allocator that must be used by STL containers.

base Objects that are dynamically allocated during VM execution must inherit from this class.

center Main entry point for the allocator.

chunk Abstraction for a memory block that contains a certain number of objects of the same size.

chunkgroup Groups several `chunk`'s for the same object size.

pool Groups several `chunkgroup`'s for allocating objects of any size.

stat For statistical purposes.

thread This allows threads to retrieve their private `pool`.

1.6 queue

This directory implements different versions of queues and priority queues. Queues may use `mutexes` for dealing with data races, use lock-free mechanisms or may be unsafe. They may also use double linked lists of single linked lists. Optionally, queues may be intrusive or not. Most priority queues use a binary heap.

1.7 external

This directory implements all the external functions that can be used by meld programs. Note that they are organized by files, depending on the external function purpose.

1.8 thread

This thread contains the different threaded schedulers. They allow running the VM on multicore architectures.

static Implements threaded execution using work stealing for load balancing.

prio Same as before, except nodes can have priority of execution.

There are some other files inside this directory, but they are somewhat abandoned.

1.9 .

meld.cpp The main VM program. Launches programs with certain schedulers.

print.cpp Prints the content of byte-code.

predicates.cpp Prints the predicates of byte-code.

simulator.cpp Runs the VM with the blinky blocks simulator.

server.cpp Runs a VM webserver that connects with a Javascript program (for visualization purposes).

2 Adding external functions

To add a new external function we need to declare the function in the compiler and implement it in the VM. The VM recognizes a set of N external functions, where each function has an unique ID between 0 and N. Both compiler and VM need to agree on each function ID.

- Edit `external.lisp` and declare new function at the end of the file.
- Implement function in directory `external/`.
- Edit `vm/external.cpp` and add a new `register_external_function` at the end.

Please see examples of function implementations to understand how to get arguments and how to return new values.

3 Adding new action facts

Adding new action facts is similar to adding a new external function:

- Edit `models/parallel.lisp` in the compiler. Add new `deftuple` at the end. The compiler will take all the predicate definition in this file and add it to each program. Note that each predicate has a `predicate_id` inside the VM that starts at 0 and ends at $N - 1$, where N is the number of predicates in the program. The predicates in that Lisp file will have the first IDs, in the same order as they appear in the file. For example, the first predicate has ID 0 and is called `_init`.
- Now we know which predicate ID our new action will have. Edit `vm/program.hpp` and add a new constant for the action you want. The examples are very clear.
- Every time an action is derived, the VM knows immediately that it is an action since that information is included in the byte-code. The function `void machine::run_action` is run in order to "run" the action. Please add a new `switch` case for your intended action using the constant you defined before.

4 Node Implementation

Each node is represented using an object of the class `db::node`. Each node has a map from predicate ID to a `db::trie`. The trie stores facts by using each tuple argument as the prefix in order to share tuples according to common prefixes. Still, at each trie leaf we store the `vm::tuple` so we can have easy access to tuples when searching.

The tries also enable faster searching if, for example, we want to match the first argument against some constant. This is done by using a `vm::match` object.

Efficient deletion of tuple leaves is also supported since sometimes we need to remove tuples from the database.

4.1 Node ID's

Every node is represented by an ID that is not associated with the node address in memory during execution. The ID is from 0 to $N - 1$, where N is the number of nodes currently in the program.

The ID used in the Meld source may not be the same as the one used in the byte-code. The meld compiler will try to optimize the topology of the program by following route axioms so that closer nodes are stored in the same thread/machine. This is also done in order to have consistent node IDs starting at 0. Anyway, the ID used in the source code is still available is called the translation ID.

4.2 Rule Matcher

Every node has an object that maintains the information about which rules are about to be fired. Every fact that is added or removed from the database must be registered in the matcher in order to know if a rule can now be run or is no longer applicable.

More information about this in the next section.

5 How rules are executed

The rule matching engine is mostly implemented in `vm/state.cpp`. After the scheduler returns a node with some work in it (`vm::state::run_node`), we call `gather_next_tuples` on the scheduler to retrieve the node's pending tuples. Note the pending tuples are tuples that were sent by other nodes.

Next, we go through those tuples (`mark_rules_using_local_tuples`) and mark them as new tuples. What this does is using the node's rule matcher to increase the counts of those predicates. When the count of predicate goes from 0 to 1, we go through all the rules where this predicate applies and increase the satisfied predicates for that rule. If all predicates of a rule are present then we mark that rule as applicable.

In `mark_active_rules`, we go through all applicable rules and add them to a rule priority queue that contains all rules that will be run. This priority queue allows us to prioritize the rules by using the order in which they appeared in the meld source code.

Before using the rule queue, we first use all the persistent tuples (`do_persistent_tuples`) to derive new persistent tuples through the application of persistent rules. Persistent rules are rules that use only persistent tuples. For each persistent tuple we execute the process associated with that predicate. This process goes through all the persistent rules where the predicate appears and tries to match the rest of the rule.

We handle persistent tuples as a special case since we do not want repeated derivations of the same tuple, which can happen in some specific situations. Note, however, each byte-code contains both the persistent predicate processes (use `./print` to see examples) and also non-persistent rules code. These differ because in the former we try to apply several rules at once, while in the later we apply just a single rule, from the beginning.

Inside `run_node` there is then a big `while` loop that, first retrieves the next rule in the priority queue and then runs it. While the rule runs, it has access both to the node's database but also to the temporary list of tuples (`local_tuples`) that we retrieved before. All consumed tuples in this list are marked and so we check them and decrease their counts in the rule matcher. Tuples that were derived during the execution of the rule are put in the `generated_tuples` list and are then put back into the temporary list (`local_tuples`). Rules that are no longer applicable because tuples were consumed are removed from the priority queue so they no longer get to run.

All the previous process repeats again and again, until we have no more applicable rules.

6 How instructions are executed

The execution engine is implemented in `vm/exec.cpp`. The class `vm::state` contains all the information about the execution state, including registers, current tuple, generated lists, etc.

Execution goes from instruction to instruction until a `RETURN` instruction is found. Every time we want to match some tuple in a rule, we do search on the database and set of temporary tuples. We iterate over the search results and do a subcall on the execution engine with a different tuple each time. The subcall will eventually return and depending on the result we may need to try the next tuple of the search results, because the matching failed. The level of nesting for subcalls will correspond to the number of different predicates used in a rule.

For more information about the instruction format, please read `docs/vm_format.pdf`.

7 Adding new schedulers

In order to extend the VM with new computation behavior, we need to implement a custom scheduler. For example, one can implement a sequential scheduler (just for executing programs sequentially) or implement a threaded scheduler, where we have several schedulers (one per thread) that run concurrently. The possibilities are many.

To create a new scheduler we must create a new scheduler class that inherits from `sched::base`. The base class is a virtual class because there are several methods that need to be implement in order to get custom behavior.

// extend switch in process/machine.cpp Finally, in order to instantiate the scheduler objects, we need to edit the `process/machine.cpp` constructor (inside `switch`). One may create several objects (for example, as many as the threads asked for) or just one.

7.1 Adding new scheduler type

Each scheduler is identified by a constant. In file `sched/types.hpp` you need to define a new scheduler inside `scheduler_type`.

7.2 Extending node type

Depending on the scheduler, we may also need to extend the base node type (`db::node`) because we need extra information at the node level for the scheduler to work. First thing that we need to do is to create a new class that inherits from the base node class.

Each scheduler class must implement the `static` function `create_node`. It receives as arguments the node ID and translate ID and creates a new node (from the custom class). The function `get_creation_function` in `process::machine` also needs to be edited to take into account this new function.

7.3 Essential methods

Next, we describe the most essential methods that need to be written in order to get the correct scheduling behavior.

find_scheduler This method must return the scheduler responsible for a specific node.

new_work A new work (tuple) is being sent to some node. This will be called when a node sends a tuple to another node in the same "scheduler" (this is found through `find_scheduler`).

new_work_delay Same as before, but the work should be delayed before it is used.

new_agg A new aggregate tuple has been created. Essentially this should work as `new_work`.

new_work_other A new tuple has been sent to a node in some other scheduler. When `find_scheduler` returns a different scheduler than the calling scheduler, this method is called instead. Note that this method is called on the sending scheduler, not the target scheduler.

new_work_remote This is called when there is some work on a node that is located in a remote location.

gather_next_tuples This method must return all tuples that are available for computation at a given node owned by the scheduler.

gather_active_tuples This method can be called during execution, it should return any tuples of a given predicate of a node, if any exist.

init This method is called at the beginning of the execution.

end This is called at the end.

get_work This function must return either `NULL` when there is no more work to be done or a node pointer if that node has some work to be done. Expect `gather_next_tuples` soon after `get_work`.

7.4 Examples

Many examples of custom schedulers are available in the VM:

sched/serial For sequential execution.

sched/serial_ui Sequential execution with user interface communication.

sched/sim For use with blinky blocks simulator.

thread/static Threaded execution.

thread/prio For threaded execution with priorities.

8 How to Call the VM

In order to call the virtual machine one must first instantiate a `process::machine` object with the program file, number of threads and scheduler type as arguments. After that we need to call `start()` on that object so that the machine starts executing. `start()` will only return when the computation is done.

Different types of exceptions may be thrown, see `interface.cpp` for an example.