PNTM – Integration of Petri Nets and Transactional Memory

Weiyi Wu, Yao Zhang, Shengyuan Wang, and Yuan Dong

Department of Computer Science and Technology , Tsinghua University, Beijing, 100084, China
w1w2y3@gmail.com wwssyy@tsinghua.edu.cn

PNTM demonstrates a new concurrent programming model, providing explicit concurrency among cooperative transactions with correctness. It integrates a special Petri net and transactional memory, and improves the performance of transactions by decreasing the rate of conflicts. The GUI part of PNTM environment is based on PNK [1], the compiler is a modified GJC [2], and the runtime is based on DSTM2 [3].

Editor The IDE provides a simple GUI with a code editor and a net editor. The editor for Petri net system is modified from PNK. All elements can have extra fields and be edited visually. The extra fields such as resources in places and code in transitions all correspond special variables and functions in the code, as in Table 1.

Table 1. Extra fields in Petri nets' elements and corresponding elements in code

code in transition	`petrinet` function name
resource in place	`resource` variable
inscription in arc	`resource` variable

Virtual Machine The code editor can compile code along with Petri nets. Before compilation, the Petri nets are interpreted to internal representation for static check. In order to guarantee correctness at the level of Petri nets, the Petri net must meet constraints below:

1. All resources must have different names.
2. One resource should appear at no more than one place at any time.
3. The input arcs to one transition should have no common resource.
4. The output arcs from one transition should have no common resource. Besides, the resources in output arcs must be the subset of resources in input arcs.

After checking the correctness at the level of Petri nets, the editor will append a piece of code for building Petri Net simulator at runtime. Hence the simulator is created and initialized at runtime. The runtime provide 5 APIs as below:

1. `AddTransition`(transition, code) adds transition.
2. `AddPlace`(place, resource) adds place.
3. `AddArc`(source, target, inscription) adds arc.
4. `Start()` starts simulation.
5. `Join()` terminates simulation.

The runtime is a Petri nets VM using DSTM2. The first 3 API can build up a simulator of a Petri net and the last 2 API control the simulator. The simulator allocate a DSTM2 `Thread` for every transition in the net. The `Thread`s are all waiting for notification. Only notified `Thread` can consume resources, call function and produce new resources. A global lock is used to protect all resources in order to make manipulation on resources atomic and prevent deadlock. Some optimizations accelerate the check-and-consume process. Hence the overhead is relatively low. When the transition consumes resources and is ready to fire, corresponding `petrinet` function is called using reflection. If all `global` variables protected by STM successfully commit, the transition will produce new resources. Otherwise the transition will revert all state and return consumed resources.

Compilation At the early stage of compilation, the compiler will recognize and mark new keyword `petrinet`, `resource`, `global`. The `global` variables need to transform to instances of pre-defined interfaces in order to meet requirement of DSTM2's APIs. For example, equivalent code to transformed "`global int a;`" is shown in Table 2. `AInt` refers to "**A**tomic **Int**eger".

Table 2. Equivalent code to transformed code

original code	equivalent code	pre-defined interface

`global int a;`	`AInt a =`
	`factory_AInt.create();`
		`@atomic interface AInt {`
		`int getValue ();`
		`void setValue (int value);`
		`}`
		`Factory<AInt> factory_AInt =`
		`Thread.makeFactory(AInt.class);`

After AST is built, the compiler will check the semantic correctness. The functions with `petrinet` modifier and Petri nets themselves should meet constraints below:

1. `petrinet` function must have function body.
2. `petrinet` function should have no parameter.
3. Only `resource`, `global` and local variables can be used in a `petrinet` function.
4. Only `petrinet` function can be called in transition.
5. Resources in incoming arcs of a transition must be a superset of all `resource` variable used in corresponding `petrinet` function.

In addition, all references of `global` variables and all left-values consisted of `global` variables must be transformed to proper getter and setter in order to meet requirement of DSTM2's API. Equivalent code to getter and setter is shown in Table 3.

Table 3. Equivalent code to getter and setter

transform type	original code	equivalent code
initializer	`global int a = 0;`	`...;a.setValue(0);`
reference	x = a + 1;	x = a.getValue() + 1;
left-value	a = x + 1;	a.setValue(x + 1);
self-operation	a++;	a.setValue(a.getValue() + 1);

The rest of compilation is the same with GJC.

Future Work We are making efforts to some basic performance evaluation.

References

1. INA:Integrated Net Analyzer, at http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina.
2. GJC available at : http://www.sun.com/software/communitysource/j2se
3. Maurice Herlihy, Victor Luchangco, Mark Moir, A Flexible Framework for Implementing Software Transactional Memory, In Preceedings of OOPSLA'06, Pages 253-262, 2006.