StreamExplorer

Pranav Tendulkar

Verimag, FRANCE

November 25, 2014

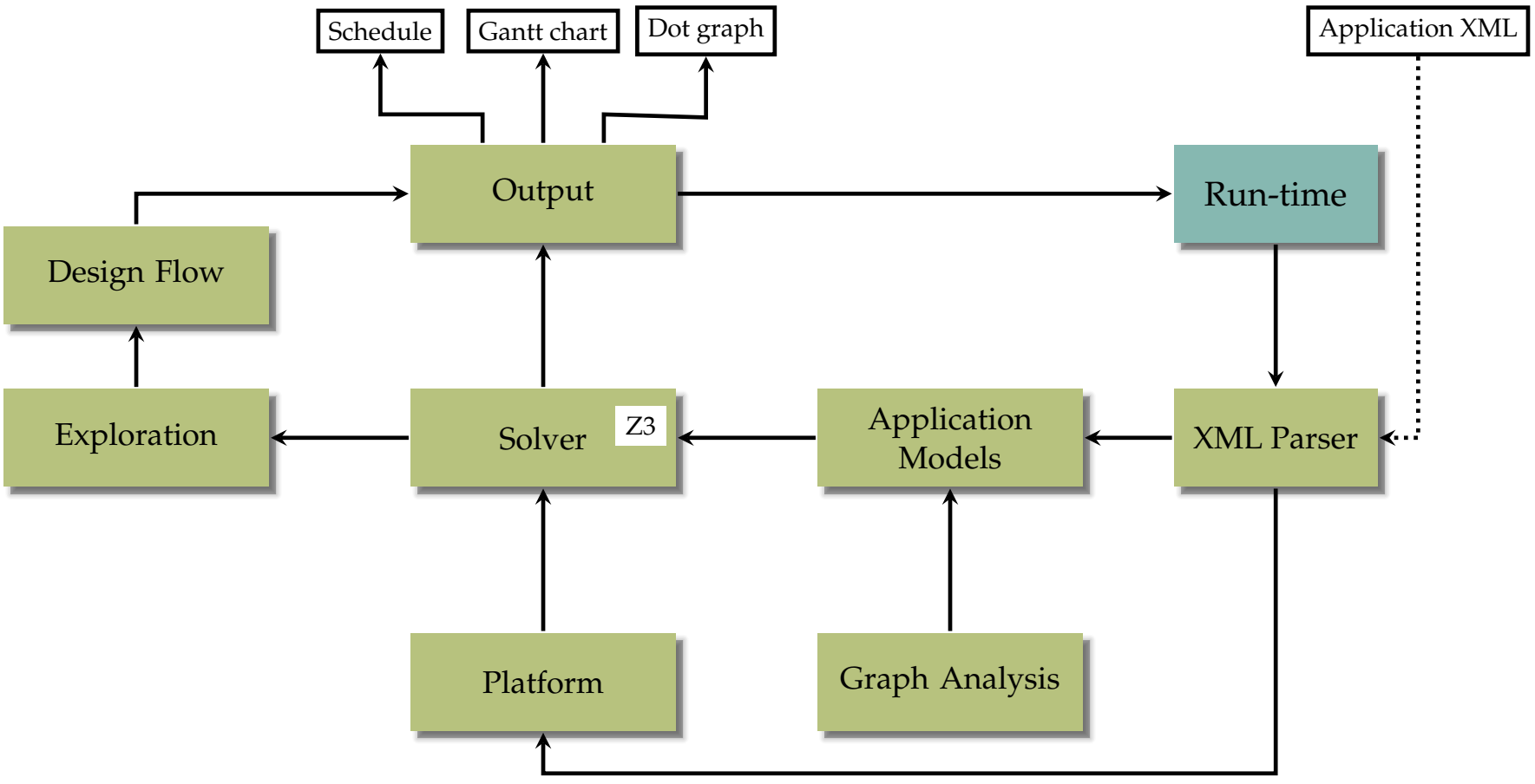
Table of Contents

STREAMEXPLORER TOOL

What is **StreamExplorer** tool?
A framework designed to

- experiment with SDF (synchronous dataflow) graphs
  - also has support for task graphs
- perform design-space exploration for mapping and scheduling
- compute scheduling using SMT solver
  - can also support other scheduling techniques
- experiment with different hardware platforms
  - Tilera TILE-64
  - Kalray MPPA-256

THE TOOL OVERVIEW



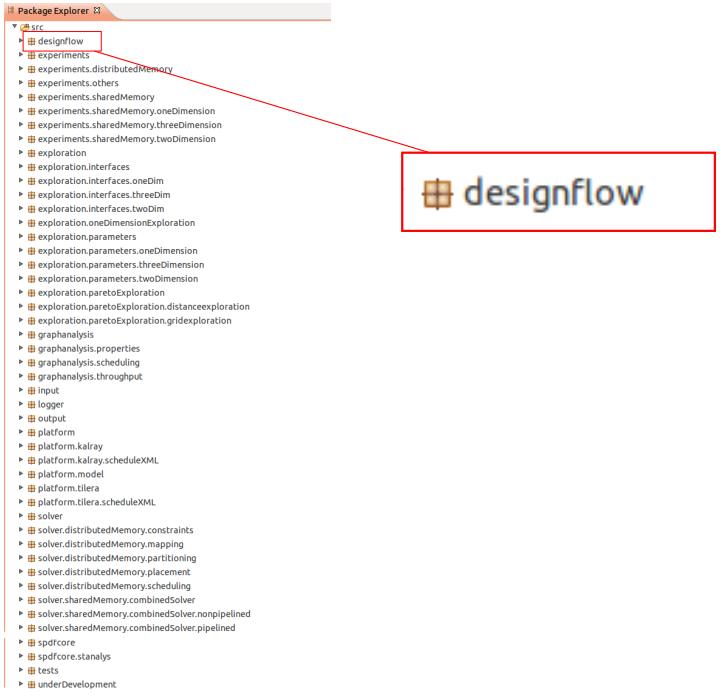
MODULE OVERVIEW

- Application models : Builds a model of application graph with actors, channels, ports etc.
- Platform: Hardware model containing information about the hardware parameters
- XML Parser : Parses XML files and build application model or hardware model
- Graph Analysis : Performs different analysis on graph models

MODULE OVERVIEW

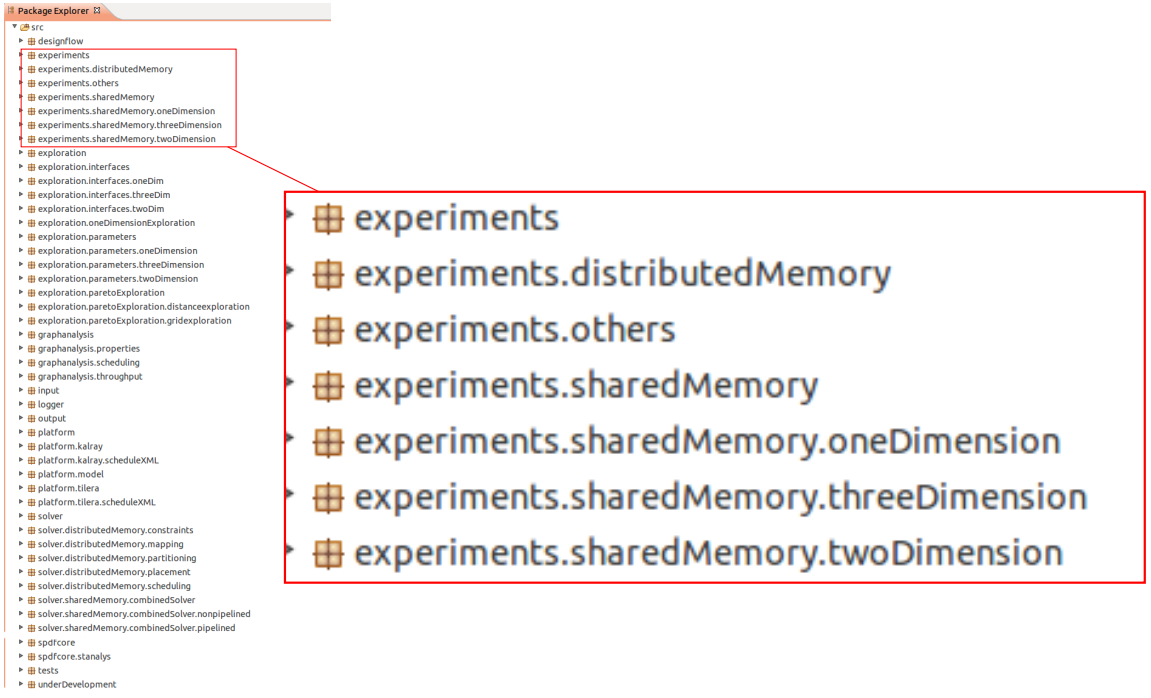
- Solver : Number of solvers for different problems like scheduling, mapping, partitioning etc.
- Output : Generate Dot Graphs, Gantt charts, schedules etc.
- Exploration : Design space exploration for different problems
- Design Flow : A three-step approach for mapping and scheduling applications on distributed memory processors.
- Run-time : C++ code running on hardware platform to execute schedules produced by the StreamExplorer

DIRECTORY STRUCTURE



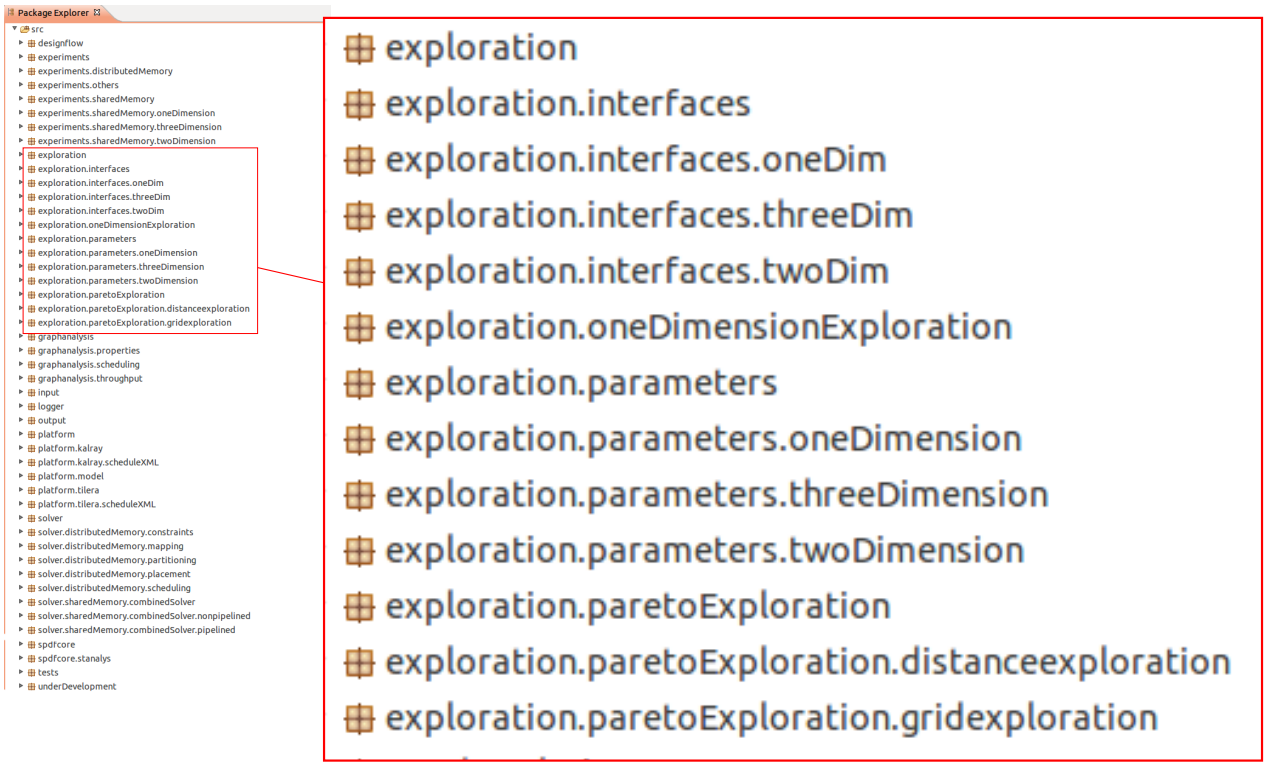
- contains class to support non-pipelined and pipelined 3-stage design flow

DIRECTORY STRUCTURE



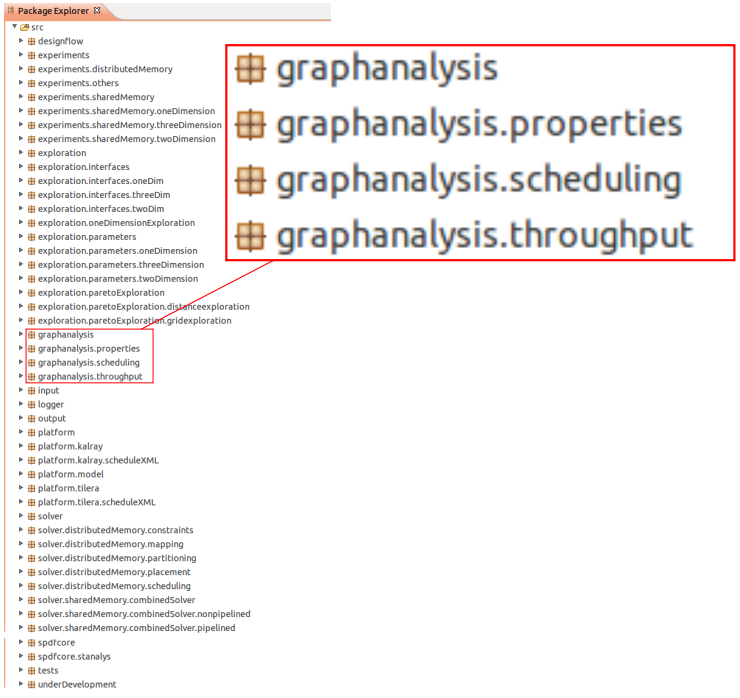
- Perform different experiments, explorations by calling main function in these classes. Shared memory, distributed memory experiments one, two, three dimension explorations. All the experiments reside here.

DIRECTORY STRUCTURE



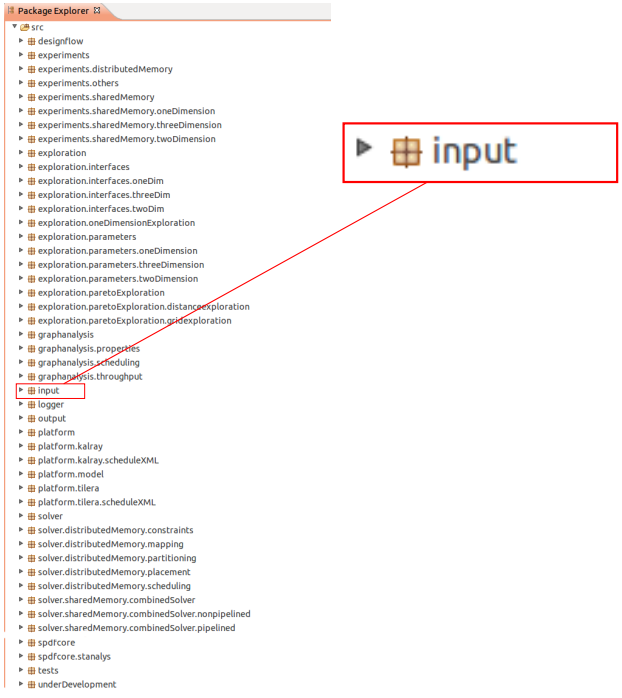
- Classes to support Design space exploration algorithms. Also contains interfaces to be implemented by different solvers and parameters which defines how the solver will be checked for different costs. It also contains exploration algorithm such as grid-based exploration.

DIRECTORY STRUCTURE



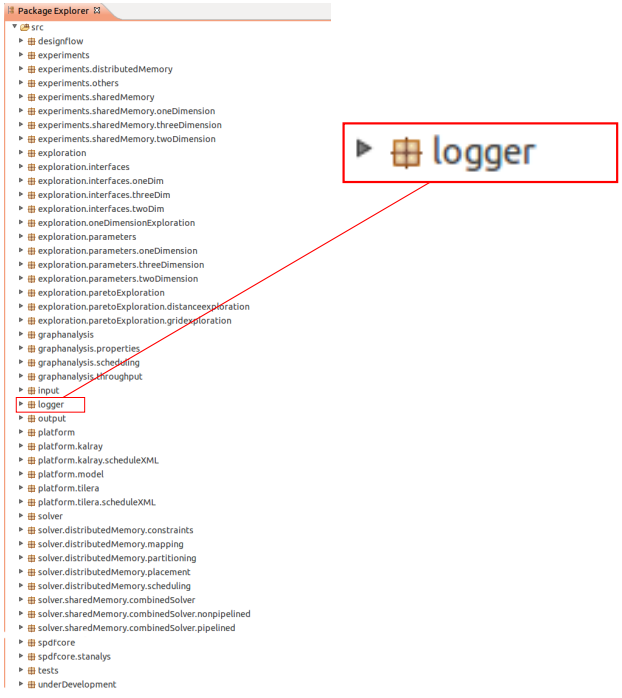
- Perform different analysis on the application graphs. Also contains classes to calculate maximum throughput.

DIRECTORY STRUCTURE



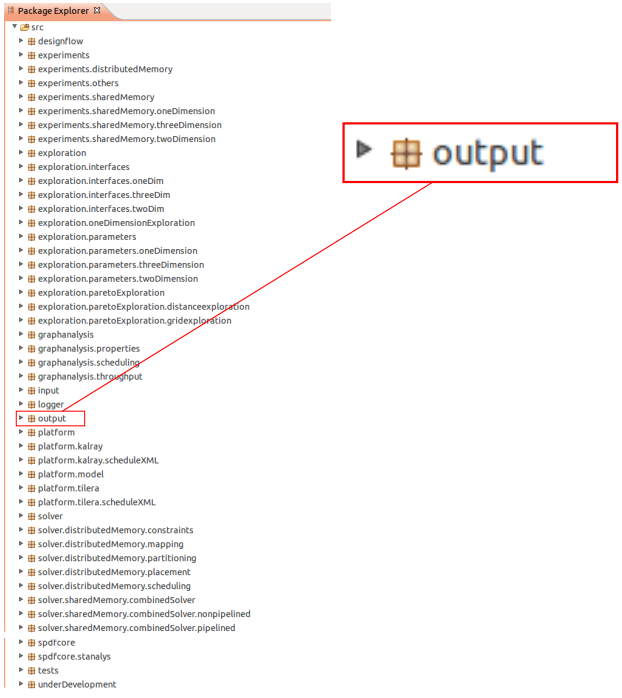
- contains classes to convert the input to the StreamExplorer to our own format. For example XML parser to application graph or platform model.

DIRECTORY STRUCTURE



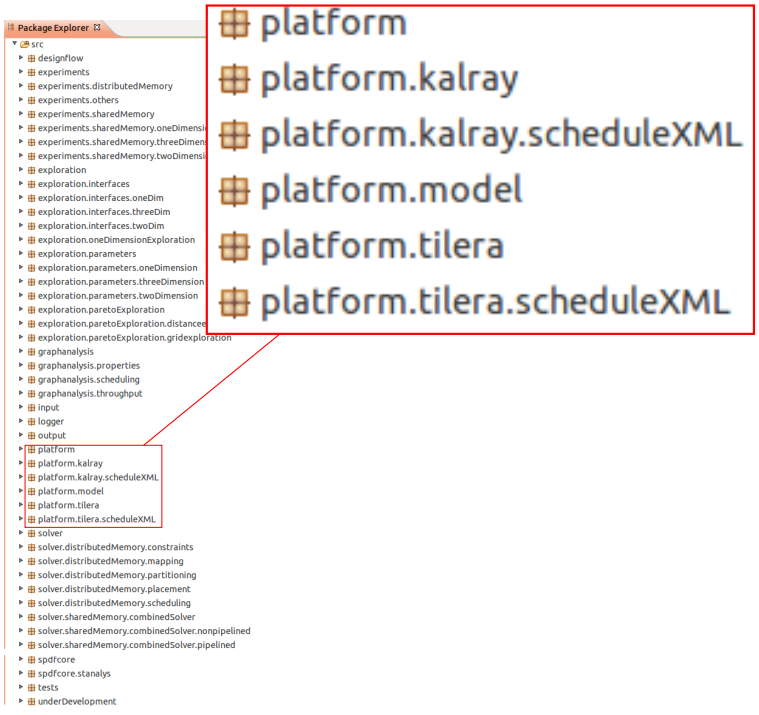
- Logs the console output to a file

DIRECTORY STRUCTURE



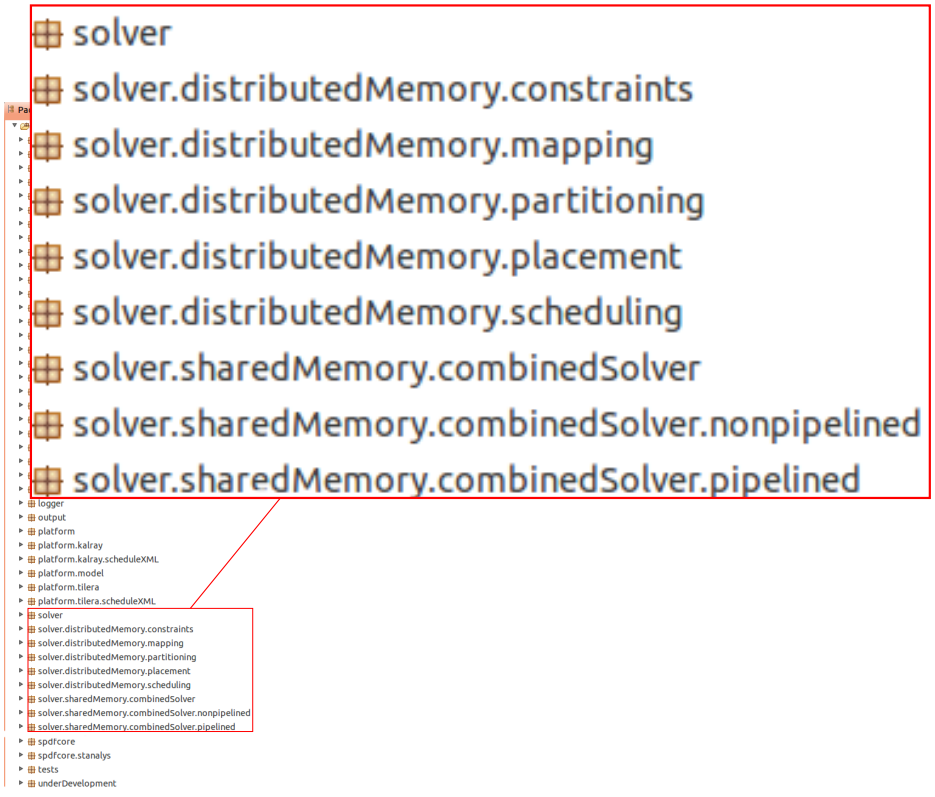
- Generate different outputs such as application XML, Gantt Charts, Dot Graphs.

DIRECTORY STRUCTURE



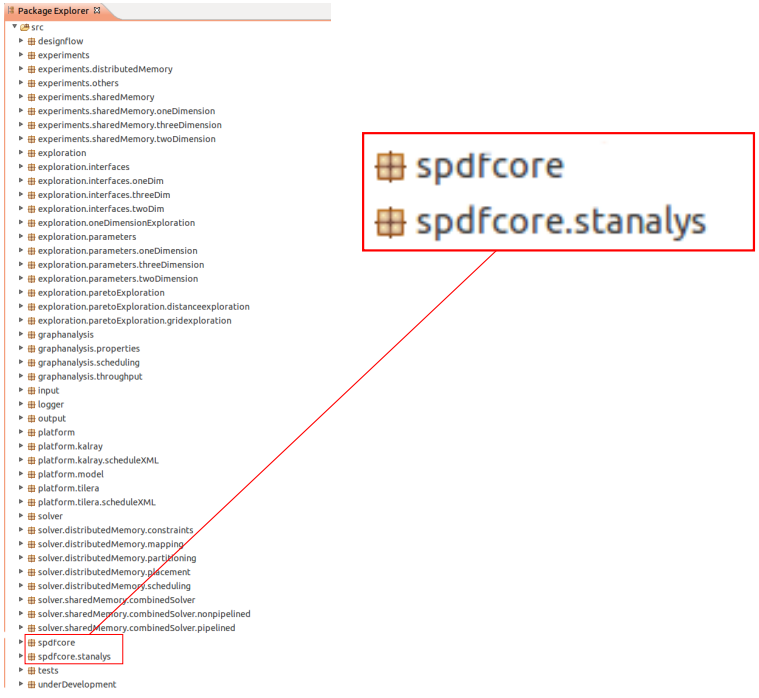
- Platform specific classes, to generate schedule XML, parse logs, generate platform model etc.

DIRECTORY STRUCTURE



- Solvers integrated with Z3 SMT solver for different problems such as scheduling, mapping, partitioning etc. for shared and distributed memory architectures.

DIRECTORY STRUCTURE



- Classes containing graph models and its components along with different analysis performed on them.

Overview

DOWNLOAD LOCATIONS

Development Repositories

- **StreamExplorer**
  svn+ssh://<USERNAME>@chaillol.imag.fr/svn/pedf/spdf-compiler-integrated-z3/trunk/
- **Kalray MPPA-256 Runtime**
  svn+ssh://<USERNAME>@chaillol.imag.fr/svn/pedf/sdf-framework/trunk/kalray/runtime
- **Tilera TILE-64 Runtime**
  svn+ssh://<USERNAME>@chaillol.imag.fr/svn/pedf/sdf-framework/trunk/tilera/runtime

DOWNLOAD LOCATIONS

Stable Version on Verimag Network

- **StreamExplorer**

/usr/local/soft/Dataflow_tools/StreamExplorer/streamExplorer.tar.gz

- **Kalray MPPA-256 Runtime**

/usr/local/soft/Dataflow_tools/Runtime/kalray/runtime.tar.gz

- **Tilera TILE-64 Runtime**

/usr/local/soft/Dataflow_tools/Runtime/tilera/runtime.tar.gz

DOWNLOAD LOCATIONS

Stable Version on web

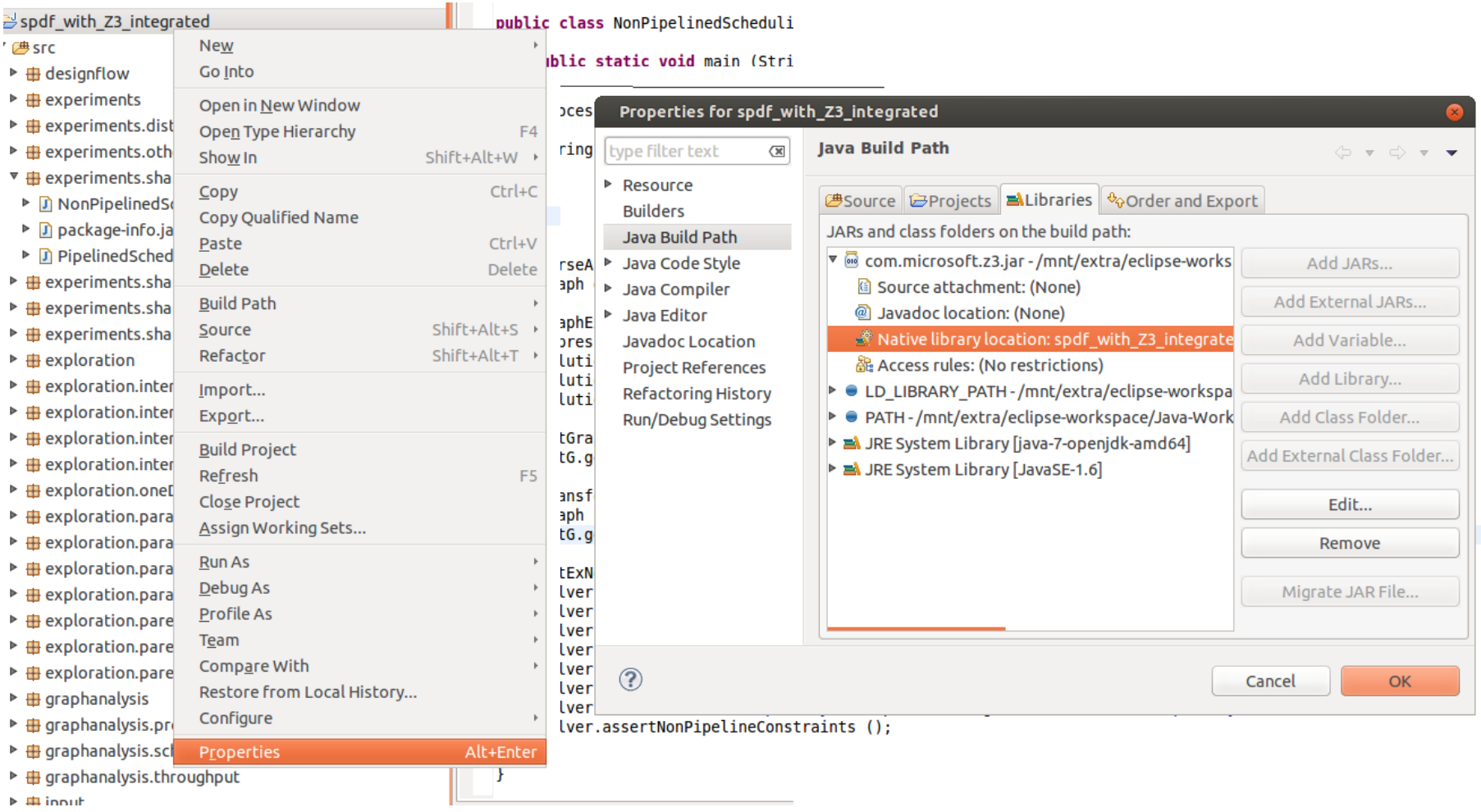
- **StreamExplorer**
  http://www-verimag.imag.fr/ tendulka/streamexplorer/streamExplorer.tar.gz

INSTALLATION

- Needs Java version 1.7
  (**Note**: if you have version 1.6 only the JPEG generation of dot graph will not compile)
- Download the source code and add to a newly created java project
- Adding to Eclipse :
  - the JAR file is located in "dep/Z3Lib/<32-bit> or <64-bit>" directory of the source code
  - make sure to add Z3 JAR in project properties (figure shown in next slide).
  - set the native library location to dep/Z3Lib/ <32-bit> or <64-bit>
  - Also make sure *LD_LIBRARY_PATH* points to the same directory[1]

[1] Note : set *LD_LIBRARY_PATH* in shell and execute eclipse in the same shell.

INSTALLATION - SCREENSHOTS



Overview

STARTING POINT

- Compile the source code from command line
  - javac -sourcepath src/ -d bin/ $(find src/ -name *.java) -classpath $PWD/dep/Z3Lib/64-bit/com.microsoft.z3.jar
- Running from command line
  - java -Djava.library.path=$PWD/dep/Z3Lib/64-bit -classpath $PWD/dep/Z3Lib/64-bit/com.microsoft.z3.jar:$PWD/bin:$PWD/dep/Z3Lib/64-bit tests.TestCompiler
- Compile the source code from eclipse
  - Just press <Ctrl> B
- Running from eclipse
  - Just open the java file and <Ctrl> F-11

STARTING POINT

To check a few classes with limited test cases :

- Run basic tests on the Compiler
  - *tests.TestCompiler.java*

User documentation for StreamExplorer and runtime resides in their respective *doc* directories.

Overview

Tilera TILE-64 Platform Quick Info

- 64-core processor (not all can be used)
- Cache coherence between all the cores
- Has virtual memory
- Library has fast synchronization (lock, barrier) mechanisms and atomic operations
- SIMD instructions for instruction-level parallelism (unused in our work)
- Dedicated networks for fast data communication (unused in our work)
- Running executable on the platform (62 processors used)
  - taskset -c 0-61 <application name>
- We configure one processor dedicated for Linux, one for I/O

EXPERIMENTS WITH TILERA

Design Space Exploration:

- Non-pipelined scheduling
  - Latency vs Processors
  - Latency vs Processors vs Buffer Size
- Pipelined scheduling
  - Period vs Processors

NON-PIPELINED SCHEDULING

Latency vs Processors:
Class :
experiments.sharedMemory.twoDimension.LatProcExploration
Parameters :

- -localtimeout : time out per query in seconds
- -globaltimeout : global time out for exploration in seconds
- -od : output directory
- -solver (mutualExclusion) : type of solver to use
- -psym (True / False) : processor symmetry
- -gsym (True / False) : task symmetry
- -proc : upper bound on no. of processors for exploration (62 for Tilera)
- -ag : application graph XML

NON-PIPELINED SCHEDULING

Latency vs Processors

Output from Exploration :

- **exploredPoints.txt** : log of entire exploration
- **satPoints.txt** : sat points found during exploration
- **unSatPoints.txt** : unsat points found during exploration
- **timedOutPoints.txt** : points timed out during exploration
- **satPointModels.txt** : model for each SAT point given by Solver
- **paretoPoints.txt** : pareto points chosen from all the SAT points
- **scheduling.z3** : ”Z3” constraints file which can be checked with Z3 binary as : z3 -smt2 scheduling.z3
- **schedule*.XML** : schedule files generated for each Pareto point

NON-PIPELINED SCHEDULING

Latency vs Processors

"Model" is a solution produced by the solver for a SAT point.
Understanding the model:

It is a Map of string to string in Java.
It contains a variable name and assigned value.

- **xVLD_0** : start time of task VLD_0
- **cpuVLD_0** : processor on which task VLD_0 allocated
- **latency** : latency of the schedule
- **totalProc** : total number of processors used

Name prefixes of these variables are defined in *src.solver.SmtVariablePrefixes*.

NON-PIPELINED SCHEDULING

Executing schedule on Tilera TILE-64

- upload (scp) *schedule xml* to the Tilera platform (tilera-eth0)
- make clean all run-hw APPLICATION=<application name> APPLICATION_ARGS=<schedule xml>
- output :
  - **rawData.txt** : information about each task execution at every iteration
  - **defaultProfile.xml** : profiling information from execution
  - **execution.txt** : execution log

NON-PIPELINED SCHEDULING

Latency vs Processors vs Buffer Size

Class :
experiments.sharedMemory.threeDimension.LatProcBuffExploration
Extra Parameters :

- -buffer : True (Enable buffer size)

SAT model :

- **totalBuf** : total buffer size of the schedule

Possible to get individual channel buffer size.

Refer to *platform.tilera.scheduleXML.NonPipelinedScheduleXml - addChannelElements()* method for an example.

PIPELINED SCHEDULING

Period vs Processors

Class :
experiments.sharedMemory.twoDimension.PeriodProcExploration
Extra Parameters :

- -solver periodLocality
- -disablePrime True
- -buffer False

Note: Buffer analysis and exploration is not yet completely supported for pipelined scheduling.

Overview

DESIGN FLOW

Non-pipelined scheduling

Class :
experiments.distributedMemory.DesignFlowNonPipelined

- Partitioning :
  - 3D exploration : max workload, comm cost, no. of groups
  - output : exploration results, partition aware graph
  - each pareto point undergoes next two steps
- Placement
  - 1D minimization : communication distance
  - output : exploration results, group to cluster assignment
- Scheduling
  - 2D exploration : latency vs buffer size
  - output : exploration results, gantt charts, schedule XML

Executing a schedule XML on the kalray platform :

- upload schedule.xml to kalray-eth0
- make clean all run-hw APPLICATION=<app. name> APPLICATION_ARGS="<schedule.xml>"

Design Flow

Non-pipelined scheduling

Partition Step:

- Exploration output : satPoints.txt, unSatPoints.txt etc.
- paretoPoints.txt contains the solutions
  - Max Workload Per Cluster : 387228
    Communication Cost : 12384
    Number of Clusters : 3
  - Max Workload Per Cluster : 730740
    Communication Cost : 2736
    Number of Clusters : 2
  - ...
- For each pareto point, there is partition aware graph
  - partitionAwareGraph_0.dot
  - partitionAwareGraph_1.dot
  - ...

Design Flow

Non-pipelined scheduling

Placement Step:

- Distance between clusters is taken from the platform XML
- Exploration output for every Pareto point in partition step
- paretoPoints.txt contains solution found with least cost
- Model contains mapping, communication cost and distance information

Design Flow

Non-pipelined scheduling

Scheduling Step:

- Exploration output : satPoints.txt, unSatPoints.txt etc.
- three z3 files :
  - scheduling.z3 : multi-cluster scheduling with buffer size constraints
  - nonlazy.z3 : reduce latency by fixing mapping and trying to make tasks execute ASAP
  - procOptimal.z3 : reduce processor usage by fixing start times and allocating tasks to unused processors if present
- Schedule XML to execute directly on the Kalray platform
- Gantt chart for visualization

Design Flow

Pipelined scheduling

Class : experiments.distributedMemory.DesignFlowPipelined

Note: This is a work in progress.

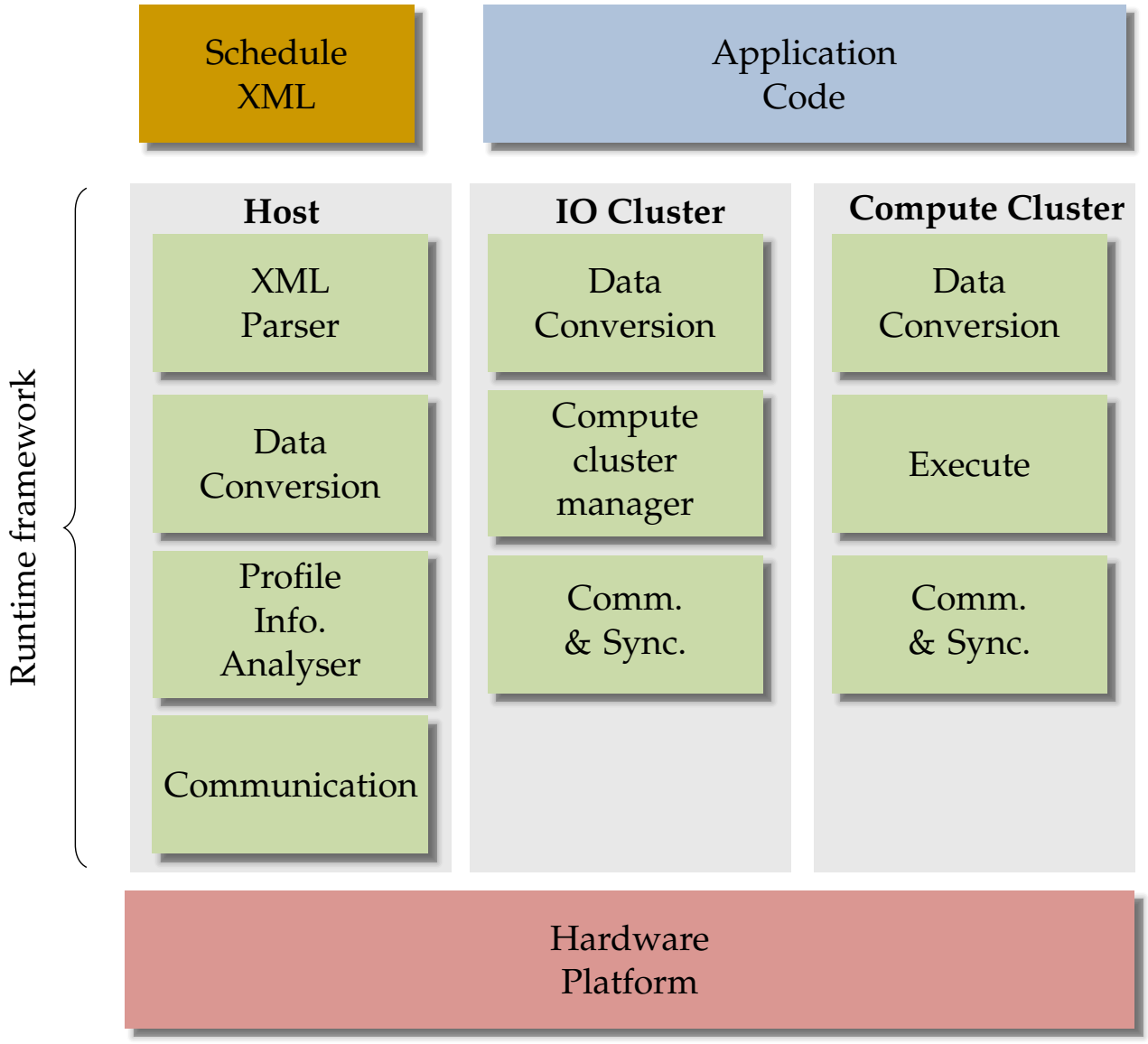
SCRIPT FOR EXPERIMENTS

All the experiments mentioned here can be tested with the help of a script in the StreamExplorer tool : *oneScript.sh*

Overview

RUNTIME

Kalray MPPA-256 platform



RUNTIME

What is **Runtime** ?
A code written in C++ for

- Execute a schedule generated by *StreamExplorer* tool
- Profile application code for the platform and provide as input for *StreamExplorer* tool
- Generate statistical information from execution of the application on the platform

Runtime on Kalray MPPA-256 and Tilera TILE-64 is similar.

MODULE OVERVIEW

- Host
  - XML parser : parses schedule XML and builds application data
  - data conversion : converts application data object to serialized plain data
  - profile info. analyser : analyses profiling information statistically
  - communication : performs data transfers with IO cluster

MODULE OVERVIEW

- IO cluster
  - data conversion : converts serialized plain data to application data object and vice-versa to upload to compute cluster
  - compute cluster manager : upload application data to compute cluster, starts & stops threads on compute cluster
  - comm. & sync. : performs communication and synchronization with compute cluster

MODULE OVERVIEW

- Compute cluster
  - data conversion : converts serialized plain data to application data object
  - execute : executes schedule on the processors inside a cluster
  - comm. & sync. : performs communication and synchronization with IO cluster

WRITING APPLICATION CODE IN C++

- Implement and extend *Application* class, and add following methods :
  - *getActorExecutor* : should return an *ActorExecutor* object corresponding to an actor
  - *initApplication* : implement application specific resource allocation / initialization
  - *exitApplication* : implement application specific resource de-allocation
- implement *ActorExecutor* for every actor in the graph and add following methods :
  - *init* : actor instance specific initialization
  - *execute* : execution of actor instance
  - *constructor* : allocate instance-specific resources
  - *destructor* : de-allocate instance-specific resources
- implement *getApplication* method to return your *Application* class object

Refer to sample application : *FifoTestApp*

RUNTIME FLOW

Kalray MPPA-256 platform

In brief the flow of the code is as follows :

1. On the host processor, the XML parser first reads the schedule XML file and builds the application data object.
2. The data is converted from application data to plain serialized data, because application data consists of different pointers and address space of host and MPPA-256 is different.
3. The serialized data is uploaded on the IO cluster. IO cluster re-builds the application data object from the received data.
4. On IO cluster, the runtime serializes only relevant actors and channels for every enabled compute cluster.

RUNTIME FLOW

Kalray MPPA-256 platform

In brief the flow of the code is as follows :

5. IO cluster spawns threads on processor 0 of every enabled compute clusters and uploads the serialized data to respective compute cluster.
6. Processor 0 of spawned compute clusters build application data from received serialized data.
7. Compute cluster initializes FIFOs for all the channels in the applications
8. It then resolves actor instances which maps an actor executor to every actor instance.
9. Then initApplication() method is called on all host, IO cluster and Compute cluster. The programmer must add application specific initialization code in this method. At this point all the FIFO channels, schedule information is ready.

RUNTIME FLOW

Kalray MPPA-256 platform

In brief the flow of the code is as follows :

10. Processor 0 of each compute cluster in execution spawns threads on its enabled processors.
11. Now all the processors execute a thread represented by method appClusterExecThread(). In this thread every processor gets its own schedule and starts executing the schedule in the order.
12. Every actor instance in the schedule is called for execution, which executes as follows
    - start all the FIFO readers
    - start all the FIFO writers
    - call actor execute() function
    - end all the FIFO readers
    - end all the FIFO writers

RUNTIME FLOW

Kalray MPPA-256 platform

In brief the flow of the code is as follows :

13. One after other the actor instances are executed in a for loop NUM_ITERATIONS times. Time is recorded if the PROFILE_APPLICATION flag is enabled during compilation.
14. After the thread execution has finished, the profiling information is sent to IO cluster and from there to host.
15. exitApplication() method is called for the application to clean-up its any of the allocated resources.
16. On host the profile data is analyzed and a profile XML file is generated from the acquired data.
17. Finally memory is released for different objects