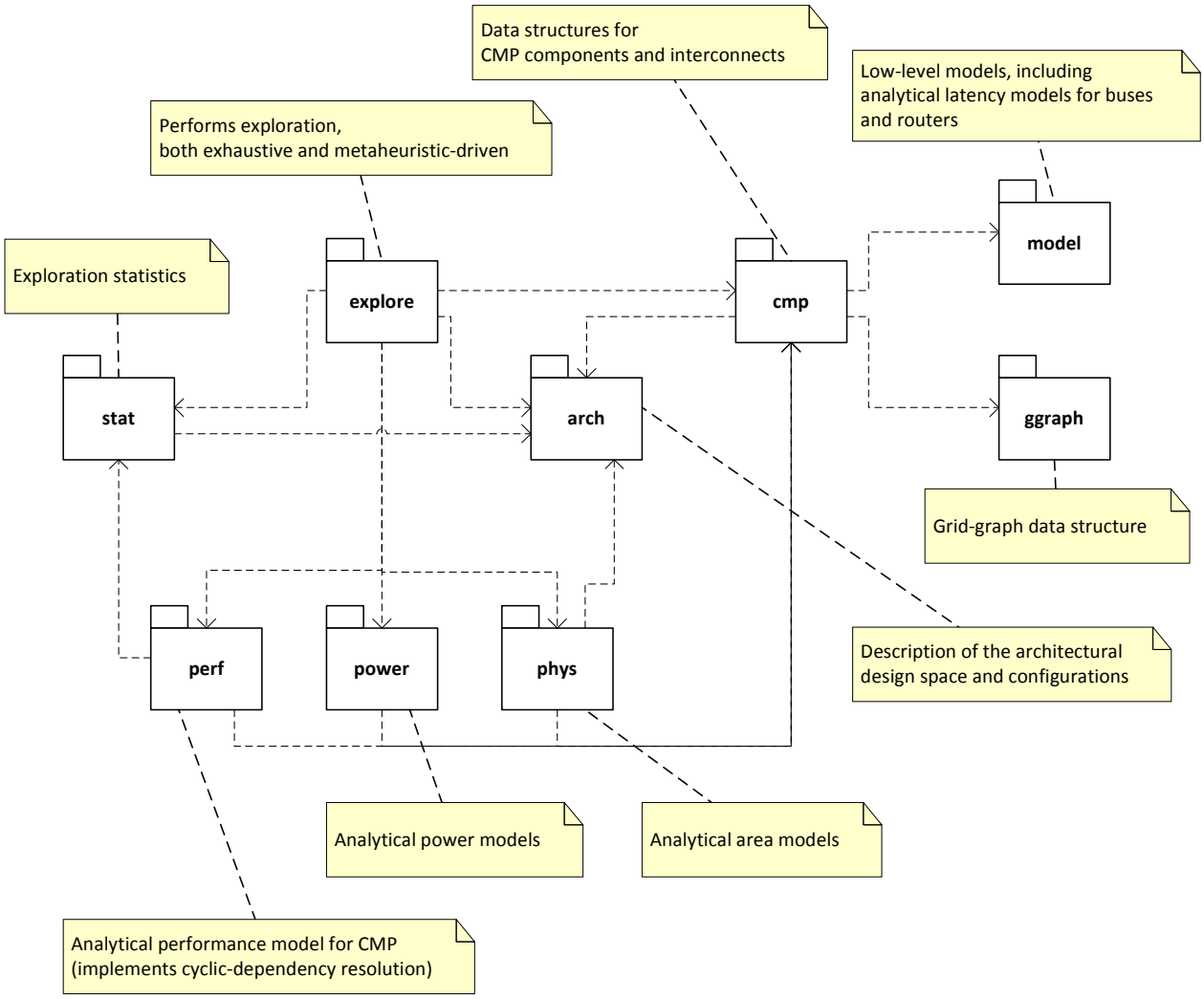
CMPexplore Developer guide

Version 0.2.1, 04 Feb 2014

Contents

1. Architectural Overview: Package Diagram and Dependencies

This section illustrates software packages of the tool and the dependencies between them.



2. Architectural Details: Functionality of the Main Classes.

This section describes the functionality of the main classes within every package.

a. `GGraph` package includes the following classes:

GVertex – is a grid vertex, used by Grid Graph.

GDEdge – is a grid directed edge, used to connect vertices of Grid Graph.

GGraph – is a Grid Graph class.

b. `Model` package includes the following classes:

BusModel – implements analytical latency model for bus interconnects (Ogras et al., TCAD'10)

RouterModel – implements analytical latency model for a single on-chip router (Ogras et al., TCAD'10), used to model the latency of router-based interconnects, i.e. meshes and rings.

Function (and derived classes) – simulate 2D mathematical functions, i.e. return the ordinate value, given the abscissa value, and depending on the function type. Functions can be read from files.

c. `Arch` package includes the following classes:

ArchConfig – represents description of a single architectural configuration. Keeps both, a set of parameters, and a string-based description.

ArchPlanner – keeps track of the current configuration during the exploration, by using the internal iterators/counters for the parameters of the design space. Generates ArchConfigs upon request, using the internal counters.

ParamIterator – is an auxiliary type used by ArchPlanner to iterate through the design space, when performing exhaustive exploration.

d. Cmp package includes the following classes:

CmpConfig – Keeps globally-accessible parameters of CMPs, i.e. fixed definitions of the design space, such as memory density (as opposed to L1Size which is explorable and hence is stored in ArchPlanner). Keeps pointer to the CMP object. For performance reasons, also owns all component objects of CMP (processors, memories, MCs), workloads, and makes them globally accessible via indices.

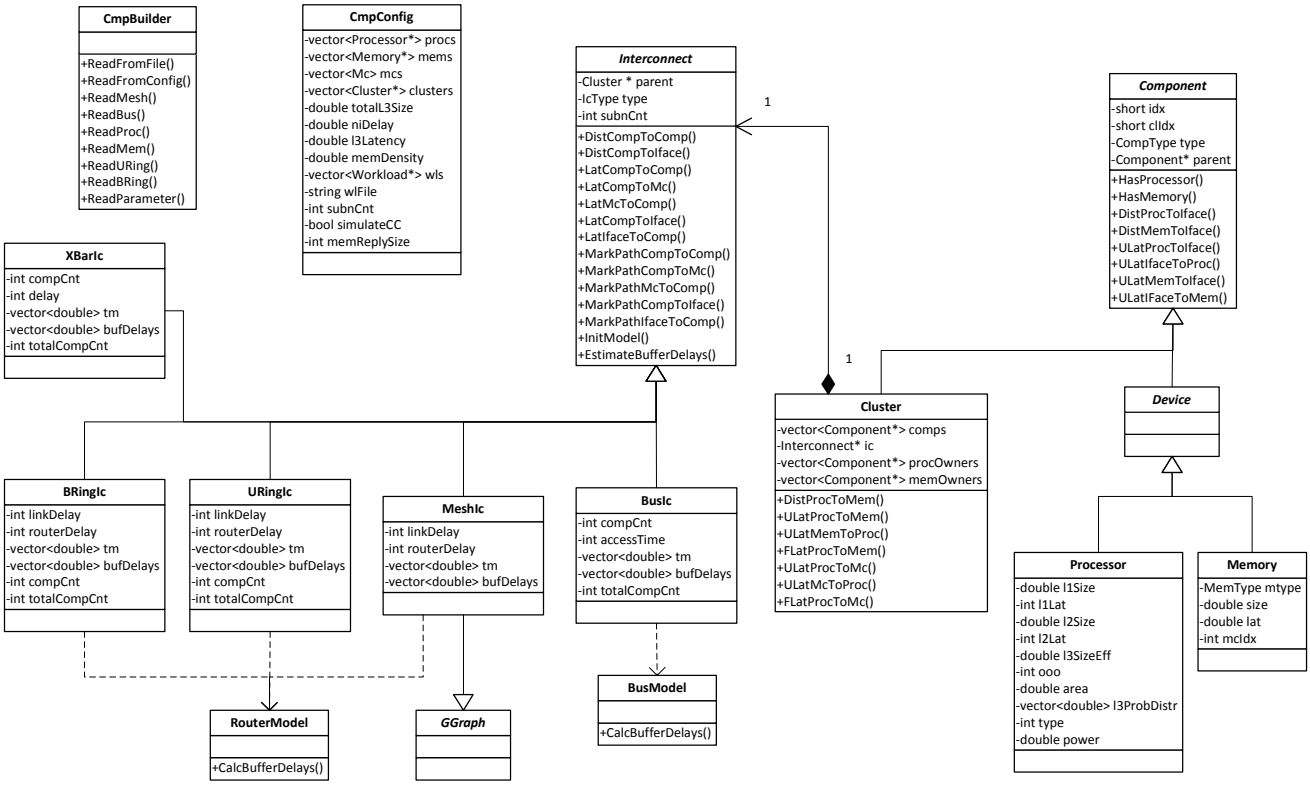
CmpBuilder – Creates a CMP object from a file description or ArchConfig.

Component, Cluster – Resemble a hierarchical data structure, implementing the *composite* pattern. Component is an interface for Devices and Clusters. Cluster is a composite object which holds a set components.

Device, Memory, Processor – Device is an interface for Memories and Processors, and latter implement corresponding components of a CMP.

Interconnect, BusIc, XBarIc, MeshIc, URingIc, BRingIc – Interconnect is an interface for CMP interconnect classes, which can be implemented as buses, crossbars, meshes, uni- and bi-directional rings.

The following diagram specifies relations between the classes of *cmp* package:



e. `Stat` package includes the following classes:

StatConfig – Represents a statistical unit, keeping exploration data for one ArchConfig. Has a set of metrics, which are the aggregate metrics for all workloads. The aggregation strategy is given by the function AggWlObjective. The ArchConfig object is owned by StatConfig.

Statistics – is a storage for StatConfigs. Provides interface for StatConfig accumulation and report generation.

f. `Perf` package includes the following classes:

CmpAnalytPerfModel – Provides methods to analytically calculate average latencies of access between cores and memories, depending on the traffic (but does not account for the cyclic dependency). In other words, this implements only L(\lambda) calculation for one iteration of the iterative performance model (see IterativePerfModel).

IterativePerfModel – Implements analytical performance model for CMP by resolving the cyclic dependency between traffic and latency. Several numerical methods are implemented: fixed-point, bisection and subgradient.

MatlabPerfModel – Used to be a writer of matlab files with nonlinear system, specifying the cyclic dependency between the traffic and latency. It was used for the NOCS'12 paper. It is outdated and is a candidate for removal.

g. `Power` package includes the following classes:

PowerModel – Implements analytical power model for CMP, based on pre-characterized power and energy values from Orion/CACTI. *NB!* For dynamic power calculation, uses performance metrics (traffic), calculated during performance modeling. This means that in order to assure correct power values, analytical performance model needs to be run first (see IterativePerfModel and the sequence of calls in code).

h. `Phys` package includes the following classes:

PhysicalModel – Implements analytical area and routability model. Note that some routability models may be obsolete and are candidates for removal.

i. `Explore` package includes the following classes:

Transform (and derived classes) – represent transformations used to generate neighbors for exploring the neighborhood of current configuration. Transformations change parameters in the ArchPlanner object, which keeps the 'current' state.

ExplConf – is a collection of parameters representing each configuration uniquely during exploration. It is somewhat similar to ArchConfig, but additionally stores the configurations metrics, such as throughput and power.

ExplEngine – is a base class for exploration strategies (engines). Implements common functions.

ExhExplore – is the engine implementing exhaustive search of the design space.

SaEngine, EoEngine, RbEngine – Implement various exploration engines, based on Simulated Annealing, Extremal Optimization and Random-Best search.

HcEngine – Implements a Hill-Climbing engine. It is outdated and is a candidate for removal.

3. An Example of Exploration

The objective of this section is to give an overview of the sequence of actions happening when a typical exploration scenario is run.

Consider an example described in the User Guide, Exploration Mode section. Simulated Annealing-based exploration is run with the following command:

```
./cmpexplore -config ./conf_demo_1.5b.txt -exp_mode sa -sa_a 0.995 -s_eff 1
-max_power 150 -print 10
```

Below we consider a high-level algorithm used to implement this exploration strategy.

Algorithm 1: Exploration with Simulated Annealing

1. Parse command line and configuration file (`./conf_demo_1.5b.txt`) and save parameters of the design space to the global `Config` object.

2. Create Simulated Annealing-based exploration engine `SaEngine`. Upon construction of `SaEngine`, all defined transformations of the design space are initialized. Start exploration by `SaEngine::Explore()`.

3. Initialize `ArchPlanner` which stores the values of exploration variables for the currently explored configuration.

4. Start typical annealing schedule.

    a. Initialize current and best configurations using `SaEngine::GetCurConfigCost()`. `ExplConf` is used to store `ArchConfig` together with its metrics.

    b. While some improvement is observed, and for several iterations at a fixed temperature, generate a neighbor by applying randomly selected transformation to the current configuration. Evaluate the neighbor with `SaEngine::GetCurConfigCost()` and decide on its acceptance. Update the current and best configurations if accepted. Scale the temperature. Note that the call to `SaEngine::GetCurConfigCost()` updates exploration statistics with the current configuration, if the latter satisfies the exploration constraints (area, power, aspect ratio).

5. Generate reports using `stats.ReportAllConfigs()` and dump configurations with `stats.DumpAllConfigs()`.

The model for analytical performance and power, used by `SaEngine::GetCurConfigCost()`, is described in the next section.

4. Overview of the Main Algorithms

The objective of this section is to give an overview of several important procedures.

ArchPlanner::GenerateNextArchConfig()

This procedure is used for exhaustive exploration. It generates next configuration, with respect to the current one, and assures that all iterators advance to the next feasible value, as given by the domain set, and that the configuration remains within the area constraint; otherwise no "next" configuration exists and zero pointer is returned. The algorithm is fully self-documented in the code.

ArchPlanner::GenerateCurrentArchConfig()

This procedure creates an `ArchConfig` object, describing the current configuration. It is defined by the values of the `ArchPlanner` iterators.

ExplEngine::GetCurConfigCost()

This procedure performs analytical modeling of the current configuration defined by `ArchPlanner`. The high-level algorithm is as follows.

1. Generate current `ArchConfig` from `ArchPlanner`.
2. Create and initialize a CMP described by `ArchConfig`.
3. Calculate area with `GetAreaOverhead()`.
4. For every workload calculate latency/throughput with `IterativePerfModel` and power with `PowerModel` (for details, see below).
5. Add configuration to statistics if user constraints are satisfied.

IterativePerfModel::RunFixedPoint()

This procedure estimates performance of CMP analytically by resolving the dependency between traffic and latency using fixed-point iteration method. The high-level algorithm is as follows.

While required precision in throughput is not reached, repeat:

1. For every processor, calculate L(λ), IPC(L) and recalculate traffic λ'(L).
2. Mark paths, i.e. propagate λ'(L) across the interconnect buffers, using XY-routing.
3. Estimate delays, i.e. run queuing model to recalculate latency L'(λ'). If this model fails, an error is reported.

IterativePerfModel::RunSubgradient()

The procedure is similar to RunFixedPoint(), apart from using the subgradient method to update the values of λ at each iteration. This assures that the failure of queuing model does not stop the procedure. The algorithm is as follows.

While required precision in throughput is not reached, repeat:

1. For every processor
    a. Calculate L(λ), IPC(L), recalculate traffic λ*(L) and L* = L*(λ*(L)).
    b. Calculate the gap between L and L*.
    c. Depending on the sign of the gap, select gradient direction (Sign).
    d. Update λ' = λ + Sign·Step(λ).
2. Decrease the size of gradient step, if the direction has changed for all dimensions.
3. Mark paths, i.e. propagate λ'(L) across the interconnect buffers, using XY-routing.
4. Estimate delays, i.e. run queuing model to recalculate latency L'(λ'). Parameter fixNegDelays is used to artificially report very high latencies in case the model fails. This allows to continue iterating the gradient algorithm.

IterativePerfModel::RunBisectionFp()

The procedure is similar to RunFixedPoint() and RunSubgradient(), apart from using the bisection method to update the values of λ at each iteration. This assures that the failure of queuing model at step 3 does not stop the procedure. The algorithm is as follows.

I. Attempt running fixed point and return if the latter terminated successfully. Otherwise:
For every processor λmax = λ(FixedPoint), λmin = 0.

II. While required precision in throughput is not reached, repeat:

1. For every processor calculate λavg = (λmax + λmin) / 2.
2. Mark paths, i.e. propagate λavg across the interconnect buffers, using XY-routing.
3. Estimate delays, i.e. run queuing model to recalculate dynamic component of L(λavg). Parameter fixNegDelays is used to artificially report very high latencies in case the model fails. This allows to continue iterating the bisection algorithm.
4. For every processor
    a. Calculate L(λavg), IPC(L), recalculate traffic λ*(L) and L* = L*(λ*(L)).
    b. Calculate the gap between L and L*.
    c. If gap < 0, then λmin = λavg, else λmax = λavg.

5. Known Issues and Limitations

The objective of this section is to describe known limitations and issues of the current implementation. The issues mentioned in this section are the candidates for future improvements in the tool.

- The notations for cache sizes are abused: 0.064Mb is used as an alias for 64Kb, although 64Kb is actually 0.0625Mb. This introduces slight imprecision in cache sizes.

- The average latency of cache access depends heavily on the cache implementation (inclusive/exclusive/…). Cache implementation determines the probabilities of accessing each of the cache hierarchy levels, together with the workload miss ratio curve. Possible models for different cache implementations can be found in ArchPlanner::GenerateCurrentArchConfig() – see the commented out section in processor defines, where the probabilities were defined previously. Currently, these probabilities are defined in the processor class: Processor::L1AccessProbability(), Processor::L2AccessProbability(), etc.

- Cache coherency mode doesn't support configurations with no L3 (since cache/directory access delay is not defined), although ArchPlanner can produce such configurations. As a work around, configurations with no L3 are simply skipped before performance estimation is run. E.g. see the "hack for avoiding configs with no L3" in ExplEngine::GetCurConfigCost().

- Certain parameters are hardcoded, e.g. router and link delay, global mesh maximum aspect ratio. For a bigger list refer to the Hidden Parameters section of the User Guide.

6. References

This section contains a list of documents which are related to some implementation aspects.

1. Analytical CMP models and methods for resolving the cyclic dependency.

   N. Nikitin, J. de San Pedro, J. Carmona and J. Cortadella.
   "Analytical performance modeling of hierarchical interconnect fabrics."
   NOCS 2012.

2. Analytical performance queuing model for buses and on-chip routers.

   U. Ogras, P. Bogdan and R. Marculescu.
   "An Analytical Approach for Network-on-Chip Performance Analysis."
   TCAD 2010.

3. Memory transactions for the simplified implementation of cache coherency protocol.

   2012-07-05_Memory_Flows_Implementation.pptx
   Supplied with the code distribution, in /doc directory.