

Visualization of computational networks

*Display of network graph parts, simulation measurement
results, and event traces*

by: *Jan Ypma*

supervisors
dr.ir. A.A. Basten
ir. S. Stuijk

09/26/2005

Table of Contents

1 Introduction.....	1
1.1 Document organization.....	1
1.2 Diagram style.....	1
1.3 Project goals.....	2
2 Computational networks.....	3
2.1 Definitions.....	3
2.2 Concurrency measures.....	3
3 Data model design.....	7
3.1 Computational network.....	7
3.2 Measurements.....	8
3.3 Event traces.....	8
3.3.1 Event file format.....	10
3.3.2 Event object creation.....	10
3.4 Designs.....	11
3.5 Summary of design choices.....	11
3.5.1 File formats.....	11
3.5.2 Event types.....	12
4 Concurrency measure visualization.....	13
4.1 Main application module.....	13
4.1.1 Graphical user interface.....	13
4.1.2 Design management.....	13
4.2 Network graph module.....	14
4.3 Node analysis module.....	15
4.4 Event trace module.....	15
4.4.1 Time compression.....	15
4.4.2 Source code integration.....	16
4.5 Bar chart module.....	17
4.6 Summary of design choices.....	17
5 Applicability of CASTviz for design exploration.....	19
5.1 Task splitting.....	19
5.2 Data splitting.....	19
5.3 Communication granularity.....	20
5.4 Data merging.....	20
5.5 Task merging.....	20
6 Conclusion.....	21
6.1 Summary of conclusions.....	21
6.2 Future tasks.....	21
Appendix A: Support libraries and programs.....	23
A.1 Scrollable desktop.....	23
A.2 Logging for java.....	23
A.3 Flexible XML object retriever.....	23
A.3.1 Class structure.....	23
A.4 'Dot' graph layout.....	24

I Introduction

Recent developments have greatly increased the available computational power of embedded computing platforms. Processing speed is up to levels in which a calculation result can no longer travel from one side of a processor die to the other within a single clock cycle. Using multiple processor cores simultaneously remains the next step in achieving higher computational power.

Combined with trends towards requiring a greater energy efficiency for computing systems, exploiting parallelism in homogeneous and heterogeneous multiprocessor systems is a very active and developing research topic. Parallel applications can be analyzed in an explicit way by describing it as a computational network, consisting of semantically and logically independent computing nodes, exchanging messages through well-defined communication channels.

CASTviz is a visualization tool for CAST[1], a computational network concurrency analysis toolkit. CAST calculates concurrency measures on a computational network, and generates event traces containing various simulation events. From these (possibly very large) amounts of data, CASTviz generates visual representations that can assist a computational network designer in overcoming bottlenecks and exploiting concurrency.

The computational networks analyzed by CAST are written in C++ using a communication framework called Yapi[2], which provides a base on which compute nodes, networks and communication channels can be defined.

CASTviz consists of various modules:

- a data access module is responsible for reading the computational network data and measures from CAST, and provides utility methods for data access to the rest of the modules.
- the main user interface module manages a single GUI window, which displays the tree of compute nodes and their calculated measures, and nested windows containing other modules.
- a node analysis module displays a list of sibling nodes for one parent node, sorted by one measurement type from a simulation. This can be used for a quick look at the nodes in the network generating the absolute worst measures.
- a graph module draws a computational network and/or its subnetworks in a graph structure, possibly using concurrency measures to accentuate bottleneck candidates or other problematic nodes by varying node size or color.
- a bar chart module generates concurrency measure bar charts for selected measures and nodes, possibly applying a linear or logarithmic rescaling.
- an event trace module renders the event traces generated by CAST in an intuitive way and draws communication arrows between them. Display space is saved by compressing the time-axis and by grouping events of similar nature.

Each module is explained in detail in its own paragraph later on.

Dependencies between the modules have been kept as low as possible. All modules logically depend on the data and main interface modules, but there are no dependencies across the other modules.

I.1 Document organization

This internship report focuses on explaining the various functions of CASTviz and the algorithms, data structures and other ideas behind them. At the end of each chapter, a short design choices summary is given, in which specific choices and possible alternatives are discussed.

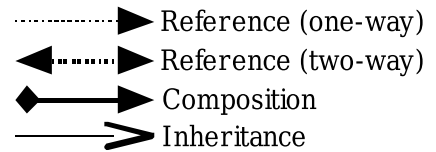
I.2 Diagram style

When explaining data structures in this document, a consistent, compact diagram style is used throughout the document. The diagram style is derived from the style used in UML Class Diagrams [3].

Data entity types (typically implemented as Java classes) are enclosed in boxes, along with important attributes (java: members) and operations (java: methods). The relations between the data entities are visualized as arrows

connecting the boxes. If the title of a box is shown in an *italic font*, the represented data entity type is abstract, i.e. no direct instances of the type will occur; only instances of object types inheriting that type.

A legend for the type of relations used in the diagrams is shown in figure 1.1. The *reference* and *composition* relation types can indicate either a relation between two objects or between one object and an object set, as is indicated by numbers alongside the arrow.



The *reference* relation indicates a one- or two-way visibility between two objects or object sets. This is typically implemented in java by using either a member or an array (List) pattern.

Figure 1.1: Legend for relations used in document diagrams

The *composition* relation is a specific kind of *reference* relation, in which a “parent” object holds, or is composed of, a set of “child” objects. The parent object is responsible for managing all child objects registered to it by composition relations. A child should only be assigned one parent by composition. In a well-designed data model, there is a tree-structured graph connecting all data objects to one or more root objects (possibly through other objects). A composition relation typically implies a reference relation pointing from the child to the parent.

The *inheritance* relation indicates a polymorphism pattern. The derived object type extends the base object type, inheriting all attributes and operations of the base type. This implies that any reference relations valid for the base type are also valid for the derived object. The inheritance pattern is natively implemented by java inheritance.

1.3 Project goals

During the internship, various design goals have driven the decisions that have been made during the creation of CASTviz. The primary goal of CASTviz is to visualize the numeric results of CAST, supporting design exploration in a logical way.

Important design goals for CASTviz include:

- To process and analyze the large amounts of numeric data that CAST generates.
- To visualize the analyzed data in a way that visual attention will be drawn to diagram regions representing the design areas that are most likely to deserve attention (e.g. bottleneck nodes). This goal is applied to all diagram types offered by CASTviz.
- Various analysis methods that will be provided by CASTviz are typically linked to a specific step in the design exploration method explained in [1].

The visualization tool is expected to assist design engineers following the design exploration method. A direct examination of this method will be performed in chapter 5, investigating how the visualization tool can assist in improving a design's concurrency.

2 Computational networks

A computational network is a collection of computing nodes and connections between them. Typically, attempts are made to optimize a network in order to achieve some efficiency, i.e. to prevent “stalled” nodes and/or overflowing connection queues.

In [1], a computational network theory is given onto which the visualization methods in this document apply. This computational network theory ascribes certain performance measures on network nodes and channels. Interpreting these measures, a system developer may then optimize network concurrency by performing a number of steps in what is called the *design exploration method*.

The various network concurrency measures and design exploration steps are explained in this chapter.

2.1 Definitions

The computational model in use in this project defines a component-based parallel application as a *computational network*, as has been detailed in chapter 3 of [1]. A computational network is represented as a directed graph, of which the nodes are referred to as *compute nodes* and the edges as *connections*.

A compute node contains a set of communication ports and an algorithmic chain of behavior, which iteratively reads from input ports, performs a calculation, and writes results into output ports. The behavior need not be in the specified order, nor is all mentioned functionality required. A node is graphically displayed in figure 2.1.

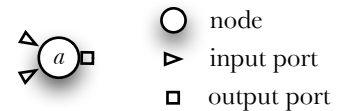


Figure 2.1: A compute node

A connection transfers communication tokens from one node to another, by implementing a fifo buffer between the nodes' ports. The buffer is assumed to have unlimited size, and the communication semantics are assumed to be blocking. This means that a write action to the connection will always return immediately (queuing a token at the end of the fifo), while a read action may block if the buffer is empty, unblocking as soon as a new write action completes.

Compute nodes can be grouped into higher-level semantic units, for reasons of reusability or application clarity. Consider figure 2.2.

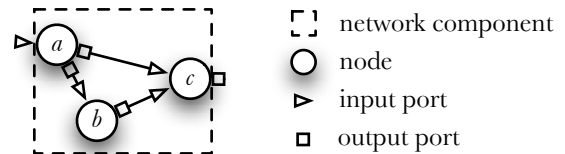


Figure 2.2: An example network component

A *network component* consists of a set of compute nodes and a set of connections between these compute nodes. Ports on the compute nodes that are not internally connected, are considered to be ports of the network component. Network components can communicate amongst each other by establishing connections on their ports.

With increasing complexity of applications, establishing a network hierarchy will increase graph clarity and can limit the complexity required at a single hierarchical component level.

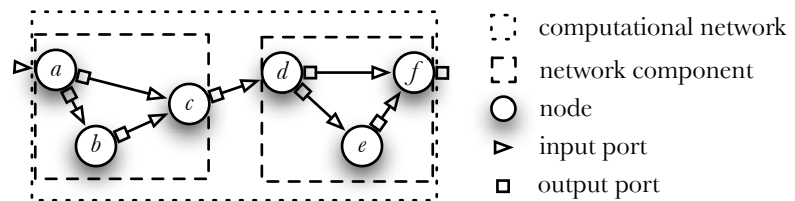


Figure 2.3: An example computational network

The *computational network* groups a set of network components and other computational networks, along with internal connections, into a top-level structure. Ports are defined in a similar way as for network components, by defining unconnected aggregated ports as the ports of the computational network. Consider figure 2.3 for a graphical example.

Being able to represent a complex computational network in an hierarchical manner, the amount of concurrency in such a system can be determined by applying the concurrency model defined in chapter 4 of [1].

2.2 Concurrency measures

In order to interpret a computing node's behavior, its activities through time are logged and recorded as specific event types. A node is assumed to be involved in three distinct types of activities:

Computation – A node involved in algorithmic calculations on data present in the node's memory. Typically blocks of computation will occur before the node performs a communication action, to exchange more data tokens to operate on.

Communication – A node engages in communication when it is either reading from or writing to a connected node. Note that write events are assumed to be instantaneous, due to the infinite communication buffer sizes. A read action may block however, generating idle time before the communication action can occur.

Idle – A node will be idling when waiting for incoming communication tokens (possibly before its first operation can start), and after its last operation has finished. A special range of idle events named *communication idle time* is defined as the set of idle events between the first communication action and the last communication read action of a node, during its run-time. Idle time is of special interest, because it indicates a node having to wait on other nodes. Reducing idle time will optimize an application.

Based on the statistics of the events' types, concurrency measures can be calculated which assist in finding network bottlenecks. The measures have been constructed to be in the range [0..1], with higher values indicating a higher degree of concurrency.

Most of these can be applied in a hierarchical manner, following the design of the computing network, enabling detailed analysis of specific potential bottlenecks in the system.

Computation load – The goal of a computing node is to get actual computing done. Although communicating with other nodes can increase productivity by allowing parallelism, performing computations should be the focus. The *computation load* of a node measures the efficiency of this balance, by dividing the total computation time by the sum of computation and communication time.

To determine the computation load of a compute network, the average of all compute nodes in the network can be taken (flattening out any hierarchy present in the network). Note that this is a simplification from [1], §4.2.1, in which the size of a subnetwork is used to calculate grouped computation loads.

Execution load – When parallelism is exploited by utilizing many nodes for computation, idle time will occur inside nodes that read from slowly-iterating other nodes. If a node reads from a connection having an empty buffer, it will enter an idle state until the connection's other side writes a token. The *execution load* of a node measures the loss due to idle time as the ratio of “useful” node time to total node execution time. It divides the sum of computation and communication time by the run-time of the node (defining run-time as the sum of computation, communication, and communication idle time).

The execution load of a compute network can be calculated by dividing the average sum of computation and communication time, by the maximum run-time of any node in the network. This way, the figure represents the average time part that the compute nodes are performing computation and communication operations.

Restart interval – To encapsulate the fact that in a system of nodes with iterative behavior (which is typical in streaming systems), quick processing and a low iteration duration are preferable, the *restart interval* measure is introduced. By defining the restart interval as $1 / \text{run-time}$, a node will have a higher restart interval as its run-time decreases.

For a compute network, the restart interval is logically defined by the lowest restart interval (highest run-time) of any node in the network.

Synchronization – As parallelism is introduced into a processing algorithm, performance is expected to increase because calculation parts can be executed simultaneously. The actual increase in performance of a parallel system with respect to its sequential counterpart with no parallelism, is encapsulated in the *synchronization* measure. To this end, the *sequential execution time* of a compute network is to be calculated, by taking the sum of all computation and communication time spent in all nodes (estimating the duration of an imaginary sequential variant of the network). The *speedup* [4] is now defined as the ratio of (computation time + communication time of the longest-running node) over the sequential execution time.

The synchronization measure is then defined as $(1 - \text{speedup})$, resulting in a figure between $-\infty$ and 1. Negative values indicate a solution slower than the sequential one, positive values indicate faster algorithms (approaching infinitely fast at a synchronization of 1).

Structure – A more explicit representation of the task parallelism of a compute network, as a result of data flows in its connectivity graph, is attempted to be represented by the *structure* measure. To this end, a compute network's complete connectivity graph is considered, discarding any hierarchy information. In the flattened network, a *path* is defined as the set of nodes which are visited exactly 1 or 2 times, when following directed connections from a network's input port to any of the network's output ports.

Collections of paths with matching in- and output ports, and only consisting of nodes present in the largest path of such a collection, are called *computational paths*. A computational path can be viewed as the set of paths belonging to the same subsequent transformations within the compute network, representing a possible data flow through the network. Effectively exploiting parallelism implies that different data flows are distinctly separated, sharing as little nodes as possible: sharing a node requires synchronization between the data flows to be taken place, hindering performance.

The structure measure enumerates this sharing of nodes. It is calculated as $1 - (\text{average computational paths through a node} / \text{number of computational paths})$. The structure will be 0 when all computational paths visit all nodes, resulting in no parallelism. With an increasing number of computational paths, and lower node sharing between paths, the figure will approach 1.

The CAST software described in [1] will calculate the given measures for a compute network, along with logged traces of the actual events which occurred during computation. The CASTviz system will visualize these results, assisted by its own extended data model.

3 Data model design

A data model for the representation of a computational network, its measurements and event traces, has been developed together with the designer of CAST. The structure of these models will be discussed in the following paragraphs.

Please note that the model described here does not fully concur with the Java implementation. For ease of explanation, some implementation-specific details are left out. These implementation details are mostly trivial, and can be partly found in the Javadoc documentation of the software source code.

3.1 Computational network

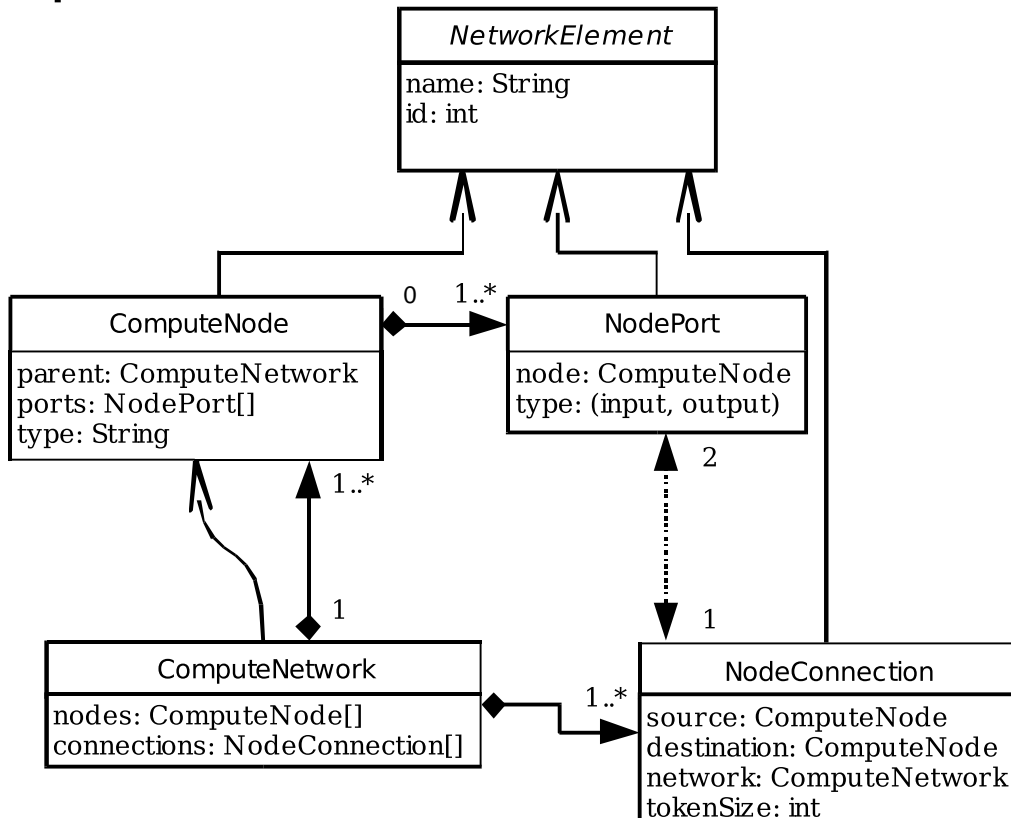


Figure 3.1: Data model for computational networks

A computational network as it is used in CAST, is hierarchically defined. The relations between the data entity types that can occur are depicted in figure 3.1.

NetworkElement – All elements that occur in a computational network are derived from this object type. For consistent references, every network element has an *id* that is unique within a design (see paragraph 3.4, Designs). Network elements also have a *name*, which is the same name that was used in the C++ Yapi model.

ComputeNode – The nodes in a computational network are of this type. Every node has a set of ports that connect the node to other nodes. The type of a node is linked with the class type that was used to construct it in the original C++ Yapi model. The parent of a node is the network that contains it. For the root *ComputeNetwork* node in a computational network design, there is no parent.

NodePort – A *NodePort* connects a node to a communication channel.

NodeConnection – The communication channel linking two nodes is represented by a *NodeConnection* object. Communication channels have a *tokenSize* attribute, which indicates the size of the type of items that can be communicated through the channel.

ComputeNetwork – (Sub)networks in a computational network design are represented by *ComputeNetwork* objects. As is shown in the model diagram, a *ComputeNetwork* inherits all *ComputeNode* attributes, and also has a set of *ComputeNode* and *NodeConnection* objects that make up the computational network. Since the contained *ComputeNode* objects could also be *ComputeNetwork*s themselves, hierarchy is achieved.

3.2 Measurements

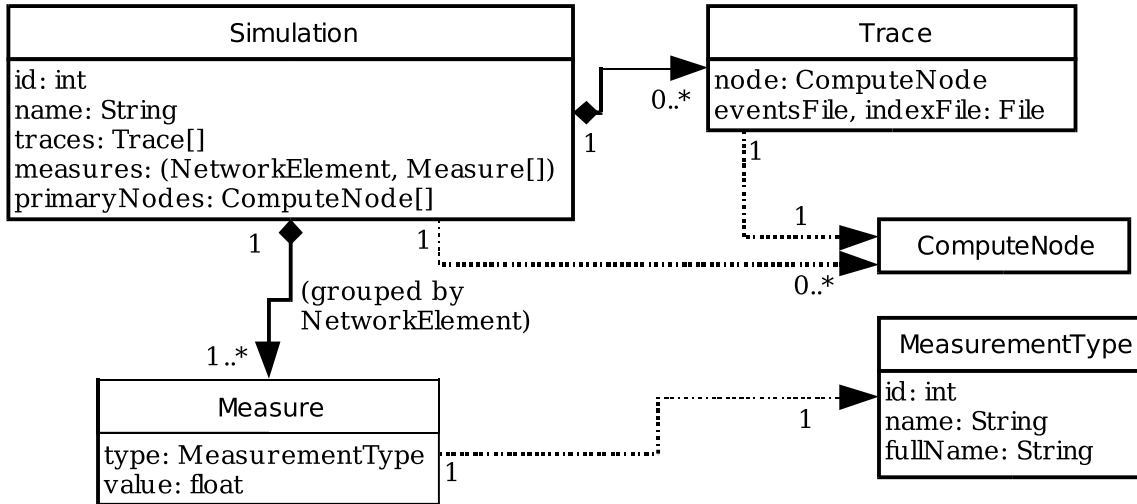


Figure 3.2: Data model for measurements

The data objects used when interpreting measurement data generated by CAST are shown in figure 3.2.

Simulation – Every simulation has a unique id and a name (typically the name of an input file that generated the calculated results). A simulation can contain a set of event traces, encapsulated by *Trace* objects.

A simulation can contain a set of *Measure* objects for each *NetworkElement* that the simulation is running on. Not all network elements need have measure results.

Measure – A calculated measurement for a specific measurement type and network element is saved in this object. The network element a *Measure* references is defined by the *NetworkElement*-based grouping in which the *Measure* objects are stored in a *Simulation*.

MeasurementType – The measurement types that can occur in a simulation are stored in *MeasurementType* objects. A measurement type has a unique id, and a short and long (full) name. Note that there is no composition relation pointing to *MeasurementType*; The measurement types are stored with a design (see paragraph 3.4, Designs).

Trace – An event trace is a comprehensive log of all actions that a specific node has performed during a simulation. The exact times that the node was calculating, communicating or idle are recorded as event types. Note that the events themselves are not shown in this diagram (see paragraph 3.3, Event traces).

3.3 Event traces

During design simulation by CAST, selected nodes can generate an event trace containing detailed information about the various node activities during the simulation. Node activities are recorded as events. From the four event types used in CASTviz, only the *InternalEvent* and *CommunicationEvent* types are native to CAST¹; the two other event types are generated by CASTviz for easier display and processing purposes.

In the CAST simulations, a global time counter is used to synchronize the various compute nodes running in parallel. Tested simulations have generated end times for this global time counter of 10^8 (or 2^{26}), so the global timer was decided upon to be stored in a variable number of bits, to allow the greatest flexibility. A Java class for working with variable-bit integers is *java.math.BigInteger*.

¹ Idle events are native to CAST, but are not logged explicitly to the files used by CASTviz.

In figure 3.3, the data model for event traces is given.

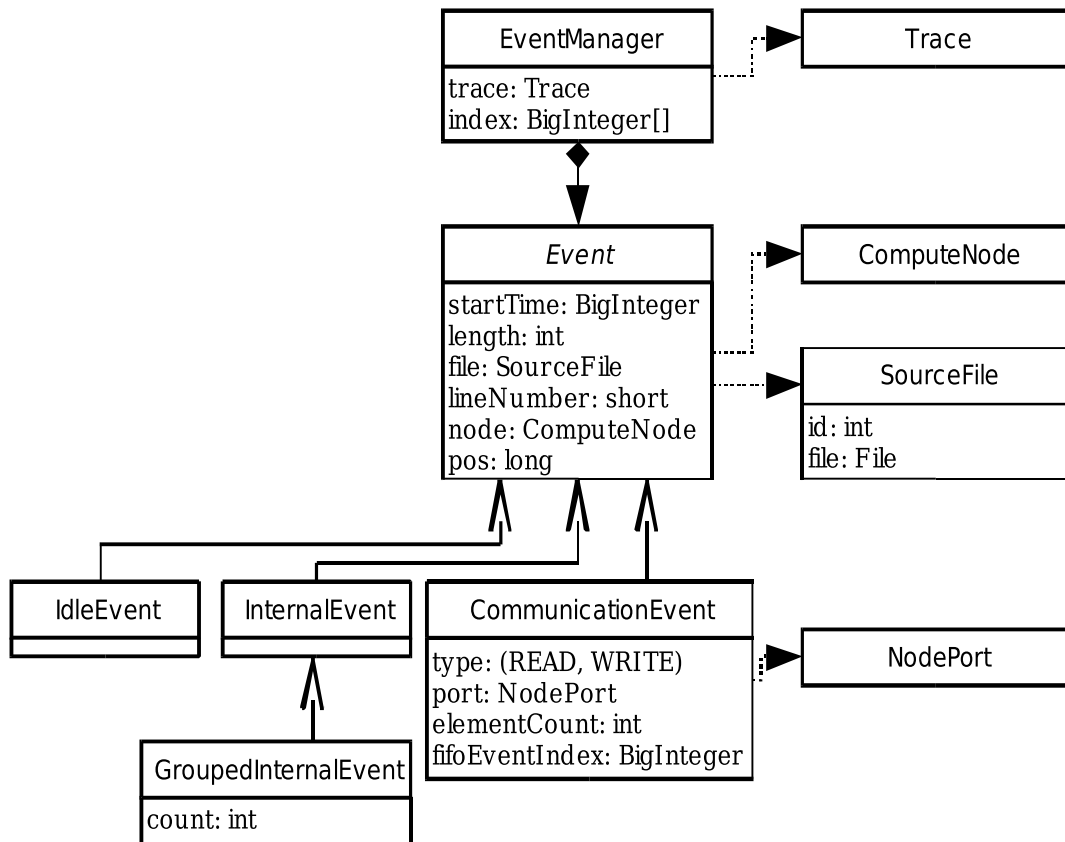


Figure 3.3: Data model for event traces

Event – All event types are derived from this abstract base type. Events have a start time and length. Events can be linked to a line number in a C++ source file, that contains the actual statement that generated the event during the CAST simulation.

SourceFile – When a design is loaded in CASTviz, the C++ source files that were used in the CAST simulation are also loaded, for fast display of source code lines generating specific events.

EventManager – The event manager provides an interface to read events from an event trace, either by global time or by direct event index. It is based on the file format explained in paragraph 3.3.1 and the event object creation strategy from paragraph 3.3.2.

InternalEvent – A node that is performing calculations generates **InternalEvent** types. Because most calculations are performed in blocks, CASTviz introduces a virtual event type **GroupedInternalEvent** that groups a continuous block of internal events.

GroupedInternalEvent – When examining an event trace, single internal events are typically not of interest. Interesting features are communication / internal event ratio and idle event occurrences. For this reason, a continuous set of **InternalEvent** occurrences are stored as a group, inside a single **GroupedInternalEvent**.

IdleEvent – A node that is not performing any actions (possibly while waiting for an incoming communication token) is said to be in *idle* state. Idle states are implicitly stored in the events file by CAST: the start time + length for event n can be less than the start time of event $n+1$; that gap is automatically detected by the **EventManager**, and an **IdleEvent** is generated to fill the gap.

CommunicationEvent – A communication action is encapsulated in a **CommunicationEvent** instance. The number of elements communicated, and the target node, are stored in the object. The **fifoEventIndex** is a

global counter, which is used to match a sending event instance to the receiving event instance on the other end of a communication connection.

3.3.1 Event file format

Event traces can grow very large. A single node can easily generate 1,000,000 or more events in a simulation run. To maintain an acceptable random-access speed in finding nodes for a given global time, an event trace is stored in two files: an event file, containing the events in a standardized binary format, and an index file, containing pointer records that count the amount of absolute time that a fixed number of events spans.

During the CAST simulation, events are written to the event file. If the global time counter has incremented by a preset amount of time $indexStep$ since the last index entry, a new index entry is made to the index file, recording the event position of

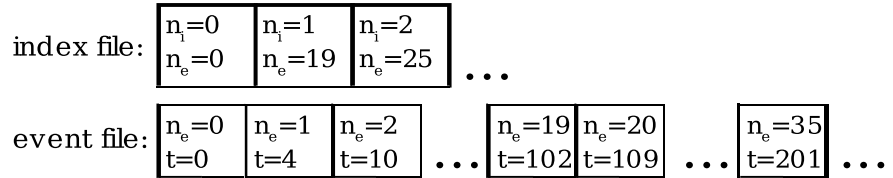


Figure 3.4: Event file format example

the event at that global time moment. An example of the relation between index and event file is given in figure 3.4. $IndexStep$ was chosen to be 100 in this example. Note that whenever the global time t in the event file crosses a multiple of 100, an entry in the index file is made at the index file position n_i of that multiple, recording the event position n_e .

The described event format allows for a fast look-up of event position n_e for a given time t , by determining the $indexStep$ -sized region of time to search from the index, and then performing a binary search algorithm on that time period.

3.3.2 Event object creation

The event manager can receive requests for events by global time, or by event position. An event request by global time can be formed into a request by event position using the index file. During this translation step, either of two situations can occur:

- An event is found that is in progress during the given global time. The request will be further dealt with as if that event was directly requested by event position.
- No event is in progress at the given global time. This implies that the node is idle at that time. An Idle event with fitting start time and duration will then be constructed and returned.

A request by event position will first retrieve the event at the given event position. If the event is non-internal, it is directly returned; otherwise, any following internal events are grouped with the first, and a grouped internal event is returned.

3.4 Designs

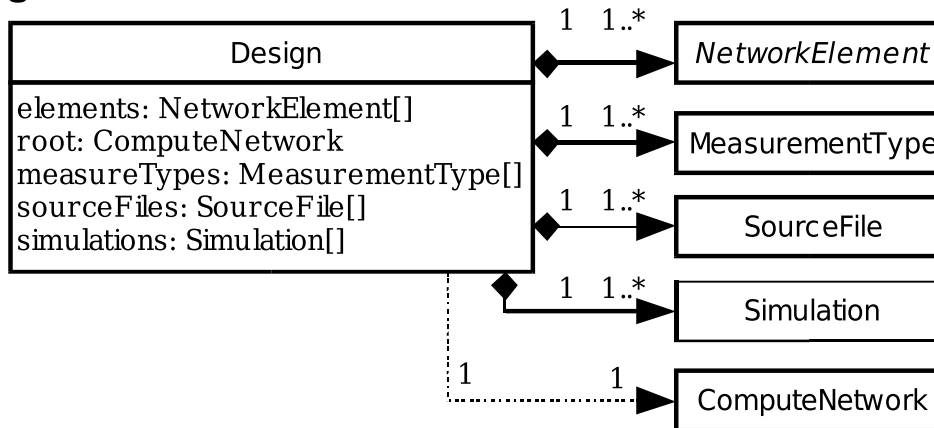


Figure 3.5: Design data model

The data model of a CAST design is given in figure 3.5. The actual data is read from various sources. The network topology and simulation results are read from a CAST xml file using the xml retrieval library described in paragraph A.3 on page 23. The events are stored in a binary set of event files. The source code is read from a set of C++ source files, referenced from the xml file.

Design – The design object functions as collector of all information associated with a design: the computational network elements, simulations, measurement types used in the simulations, and source files. There is also a pointer to the root of the computational network.

3.5 Summary of design choices

3.5.1 File formats

Because CAST is written in C++ and CASTviz is written in Java, compatible and well-defined file formats had to be developed. For data exchange between programs, three file format categories can be defined:

Flat text file – These file formats contain ASCII lines of text, in which the information is formatted in an application-specific way. Such a file is easily written and parsed, but newer file versions can break old software, because the data format is not contained in the file itself.

XML text file – This can be viewed as an extension of the above format; portability is preserved because of the text contents, but the XML formatting adds a well-defined data format to the file. XML does require a bit more processing logic though. XML is in many cases a better choice than the flat text files.

Binary file – A binary file contains a collection of records, each consisting of a fixed amount of bytes. The binary contents of the records must be very precisely known to the applications using the file. Java for example always uses Big Endian for integer storage, whereas the storage format on C++ platforms can differ. During the internship, a set of C++ routines was created to write out a binary format that is native to Java (specified in the `java.io.DataInput` interface). Binary files have the disadvantage that versioning is next to impossible; older applications generally won't understand binary files of a newer version.

Binary files can be easily written, and given an exact specification, can also be easily read. A second advantage is that because all records are of the same size, *seeking* for a specific record number within the file is very fast.

The data exchange between CAST and CASTviz consists of three categories: network topology, concurrency measures and event traces. For each, the file format that was chosen will be elaborated.

Network topology – Because the topology of a computational network in Yapi is very tree-like (due to the C++ object-oriented approach), XML is the only well-defined file format that applies. There are no advantages in the other file formats that apply here.

Concurrency measures – CAST's concurrency measures add zero or more figures to every node. This kind of data gives the choice between a binary file format and XML. For CAST, XML has been chosen, because portability and ease of use had a priority in this area.

Event traces – As described in paragraph 3.3.1, the event traces generated by CAST can become very large because every single event is logged. This means that the event file will be accessed often in a random way. Therefore, a binary format was chosen here. Additionally, because of the decoupling of event numbers and their time stamp, a binary index of the event trace file is generated along with the file.

3.5.2 Event types

Of the various event types shown in figure 3.3, only the `InternalEvent` and the `CommunicationEvent` are native to CAST. The other events are dynamically created by CASTviz to make certain tasks easier.

Grouped internal events – A typical compute node will have a pattern of first an incoming communication, then a series of calculations (resulting in `InternalEvent` instances) and then an outgoing calculation. When examining a computational network, a designer is typically looking for idle periods or periods of heavy communication.

The individual events in a calculation event group are not of much interest, and when drawing events by activity instead of by time (see paragraph 4.4.1), a lot of visual space can be gained by showing a continuous series of internal events as one grouped event.

Idle events – Events in the event file generated by CAST are time-continuous; the time a node is spending waiting for an external event is not reported. However, the event drawing and searching routines used by CASTviz expect the event “stream” to be continuous in time (this greatly simplifies various algorithms). For this purpose *idle* events are introduced whenever a gap in time is detected between two events in the CAST event file.

4 Concurrency measure visualization

The CASTviz application is split up in several application modules. The various modules are explained in this chapter.

During development, a lot of care was applied to ensure minimal dependencies between the various visualization modules. Utility methods and helper algorithms were put in the main or data module where possible. That results in an easily expendable, modular system. CASTviz can be easily expanded with another module.

4.1 Main application module

The CASTviz main application module provides multiple windows visualizing several aspects of CAST designs that have been loaded. A *multiple document interface* setup has been chosen, enabling grouping of the various application modules within a single window.

Calculations and graphing is based solely on a common file format defined by the CAST system. There is no dependency on Yapi entities within CASTviz; if the CAST simulation tool would be ported onto a different concurrency platform, its visualization counterpart will still function without modification.

4.1.1 Graphical user interface

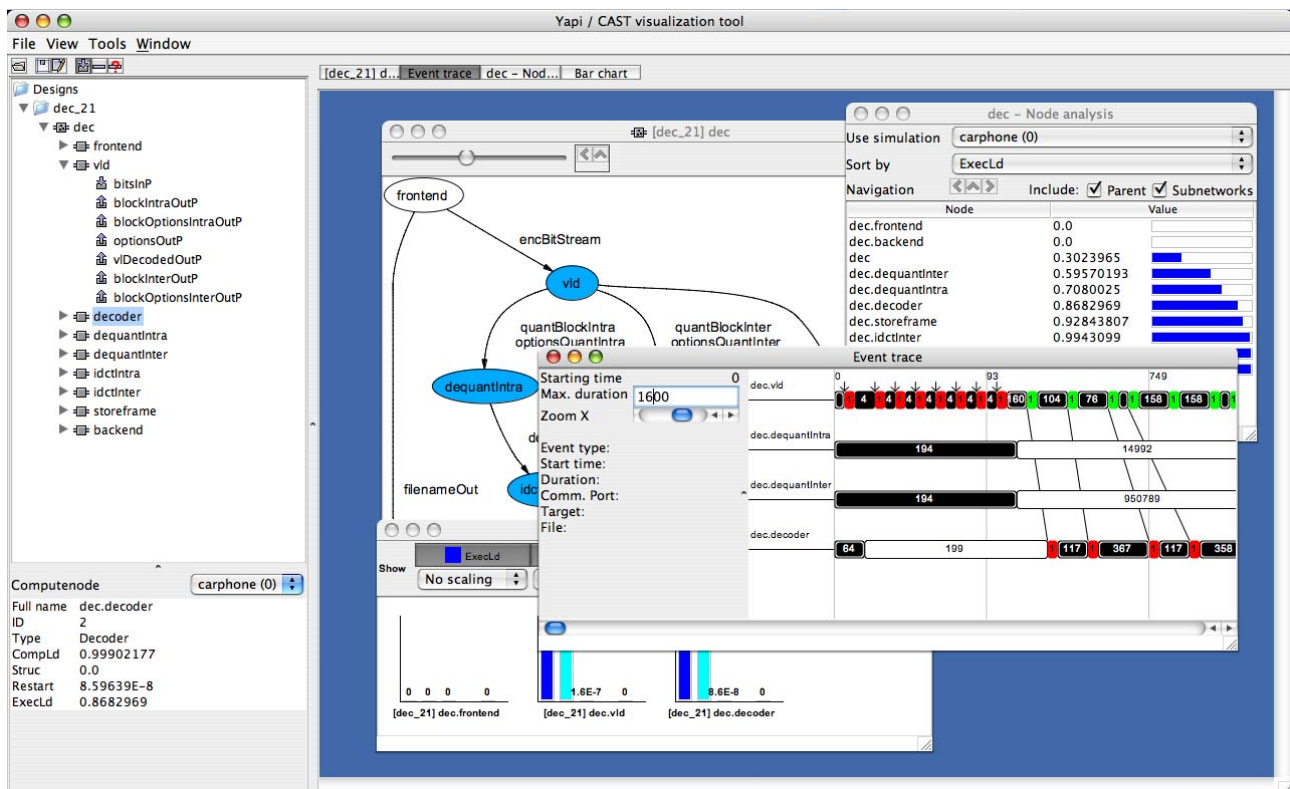


Figure 4.1: Screen shot of main application module

The main GUI window of CASTviz is vertically divided into two regions (refer to figure 4.1). A tree view on the left displays each opened designs tree structure. Beneath the tree view a property pane lists the properties and/or computational measures of the currently selected item in the tree. The right part of the main window is composed of an MDI desktop area capable of containing various windows. The other application modules typically open windows on this desktop pane.

4.1.2 Design management

Multiple designs can be mounted in the main left-side tree view. Designs can be re-loaded after a recompilation or other external events. The computational network that is described in a design is displayed hierarchically in the tree view.

When a design, compute node, or other network element is selected in the tree, the properties pane displays any static properties and/or simulation results in a list. For designs containing multiple simulation, the simulation used is indicated above the properties list. Note that selections in other windows will typically cause the main window to also select a specific network element and/or simulation, to guarantee a recognizable selection work flow.

4.2 Network graph module

The computational networks used in Yapi and CAST can in some design stages grow in size to be hundreds of nodes. A logical way to visually navigate amongst the compute nodes in a design is to display them as a hierarchical graph (see figure 4.2). CASTviz's graphing module is based on this idea, implementing the following extra features:

- Hierarchical navigation through the design is possible in two ways. Computational (sub)networks can be double-clicked to “look inside” the computational network on the network's hierarchical level, or the network can be “expanded”, to see it's contents along with the parent network. The expand function can also be applied recursively to expand a whole network.
- Compute nodes, computational networks, and simple nodes all have different base colors (compute nodes have a blue tint, networks have an orange tint, and simple nodes always are white). The base colors can be set to be influenced by simulation results, changing the size, color hue and/or color saturation of nodes or networks, according to the simulation results. This *network graph mapping* feature can quickly indicate performance bottleneck candidates.

These “problem” nodes will stand out in the graph, having a large size or a noticeable color, because of bad computation loads, execution loads, or other CAST measures.

- CAST will log a compute network with labels on all nodes, ports, and connections (this is implied by the topology structure of the underlying Yapi system). Displaying all labels in a graph would quickly clutter up display space. CASTviz will only show node labels and connection labels. Furthermore, when constructing the graph, multiple connections between two nodes will be grouped into a single graph arrow with multiple labels, to save space.
- Of course, selecting a node in the graph module will also select it in the main application module's tree view; node-specific functions are accessible from the same context menu as is used in the tree view.

The external 'dot' program is used to calculate placement of the network graph's nodes and edges. The calculated layout is re-interpreted by CASTviz, adding user interface navigation to the displayed graph. The limitations of this construction and possible enhancements are discussed in appendix A.4.

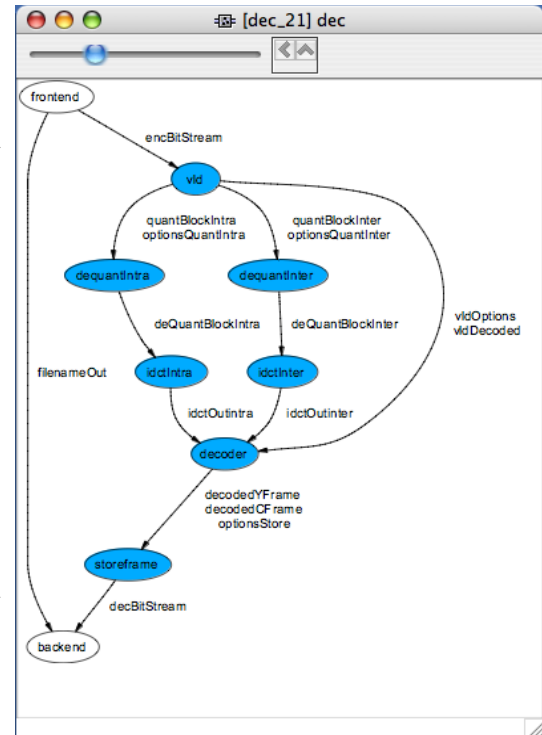


Figure 4.2: Screen shot of graph module

4.3 Node analysis module

The node analysis module displays a recursively-generated lists of nodes that are siblings of a specific parent node, sorting the list by the results of a specific measurement type from a simulation. Consider figure 4.3.

A node analysis window can be opened for the root compute network of a design, to compare all nodes in the whole design, but it can also be opened on a sub-network to concentrate on a part of the design. By selecting a specific measurement to sort on, the node analysis window can provide a more detailed view on a network region's performance.

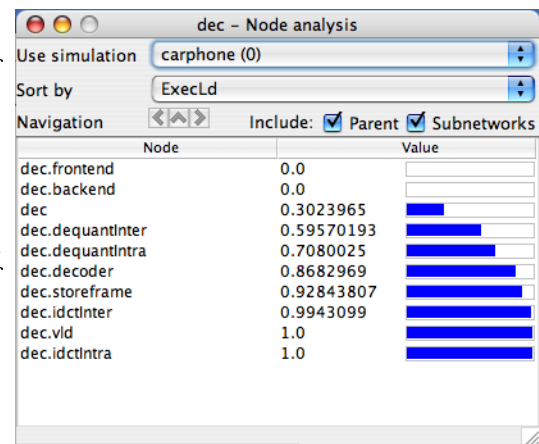


Figure 4.3: Screen shot of node analysis module

4.4 Event trace module

CAST contains a code annotation module that extends the source code of the network that is to be simulated with extra statements calculating performance measures. That feature has been extended to log all activity of selected network nodes into a binary file. CASTviz will read these files where available, rebuilding the time stream for one or more nodes, and visualizing the exact node events that occurred during simulation.

Early experiments in drawing event traces showed that keeping a linear time axis will result in huge diagrams with sparse locations of concentrated events. The idea came up to dynamically compress and expand the time zoom level, depending on the absence or presence of events. This *time compression* will automatically present event concentrations in detail, while skipping over long periods without activity.

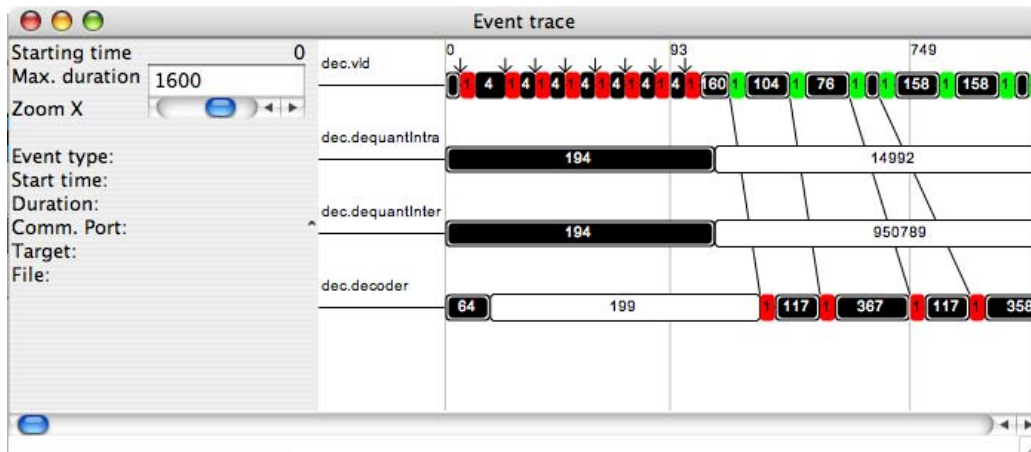


Figure 4.4: Screen shot of event trace module

An event trace in CASTviz consists of one or more nodes that have event files associated to them by CAST. CASTviz will calculate the most efficient time axis mapping for the selected nodes and their events, and communication arrows will be drawn between the events. A screen shot of the event trace module is given in figure 4.4.

4.4.1 Time compression

The algorithm involved in drawing event traces in the above manner, decides upon a monotone increasing mapping between actual timestamps and time axis locations on the screen. It does so by processing events one at a time from all candidate nodes, considering all nodes equal. The node that will advance the time pointer the *least* forward in time will be selected as to be drawn next. It should be noted that the algorithm utilizes the fact that events for a node are always successive in CASTviz (there are no gaps between events).

This is performed using the following steps:

1. Time pointer is initialized at zero.
Screen pointer is initialized at left window border.
For every node in the trace, event pointer is initialized at the first event.

2. Repeat steps 2-5 until the screen pointer is out of visible screen area, or until all event pointers are beyond the last event for their node.
3. From all event pointers, select the event(s) which have the *lowest start or end time*. If this results in multiple events with the same end time, select the event with the *highest start time* from this group. Increase the event pointer on the trace of which the event has been selected.
4. The selected event's exact start time will either coincide with the end time of a previous event, or it will be beyond any end time of events already drawn. Depending on the case, either align the start time with the screen position of the coinciding event, or set the screen position to a fixed indent beyond the last event's end time screen position. Further increase the screen position, if the time label which is to be drawn inside the event, would not fit. Repeat this for the event's end time, ensuring that the event has a non-zero width on screen (that would happen if both the event's start and end time are beyond all previous events).
5. Draw the event on the calculated screen location, and store the relation between the event's start and end time and the corresponding screen locations.

Consider figure 4.5 for an example. The algorithm is applied to the displayed events, resulting in the alphabetic ordering indicated in the picture.

Events are colored based on their event type and communication direction (internal, idle, read, write).

After drawing all visible events on screen, communication arrows are drawn for any communication events for which both write and read events are visible. If a communication event's peer is not visible, only a short directional arrow is drawn at the event.

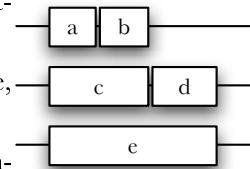


Figure 4.5: Time compression example

The diagrams created by the proposed algorithm naturally expand regions in time with many communication events, while compressing regions with large calculation chunks. This results in the viewer's attention being drawn to diagram parts with much interaction.

Although the algorithm will introduce a somewhat fluctuating time scale along the x-axis, the typically repetitive nature of the algorithms executed by compute nodes causes some leveling of the fluctuations. Care should be taken however when interpreting an event trace's time axis.

4.4.2 Source code integration

The binary format in which event traces are stored, also includes line numbers of the source lines and files that generated specific events. This information is used by CASTviz to allow a direct display of the source code fragment that is responsible for an event.

When the mouse hovers over an event that is drawn in an event trace, the corresponding source file is loaded automatically in a source code viewer, highlighting the line that generated the event. Since this CAST-generated information is not always exact, slight line differences can occur.

This feature offers a very direct way for the design engineer to identify and possibly solve design bottlenecks that are originating from a single piece of code. If at multiple points in the event trace, a bottleneck or lockup is seen, and one code fragment keeps “popping up” as the cause for that, the solution area is easily found.

4.5 Bar chart module

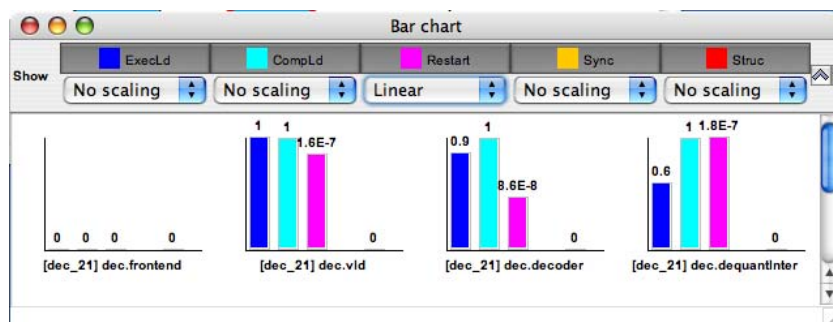


Figure 4.6: Screen shot of bar chart module

While developing the design exploration method explained in chapter 6 of [1], comparing multiple measurement types across a large number of designs showed the decision path that was followed in optimizing a design. This resulted in a requirement for automatic generation of bar charts, that show whole-design measurements like execution load, computation load, etc. on a range of design alternatives.

The user interface for bar chart creation (figure 4.6) is straightforward. An empty bar chart window is created by a menu item. Computational networks can be dragged and dropped onto the bar chart window to add a bar chart. There are also shortcuts to add subnetworks of a computational network that has already been dropped onto the window, or to add all top-level networks from all designs.

This allows to create a high-level overview of performance on the designed that are opened in CASTviz, or a lower level description of the performance in various sub-nodes of a single design, both requiring few user steps.

4.6 Summary of design choices

Several architectural and aesthetical choices have been made to optimize visualization effectiveness.

Tree view and property pane – The choice of using a single tree view / property pane combination in the user interface was made to resemble interfaces like the Windows Explorer. Users are used to seeing information in a tree view, and clicking on various nodes to see specific details on that node. Combined with the tree-like nature of Yapi-designs, a tree view is a logical choice.

Two types of hierarchical network navigation – The network graph of a multilevel network can be navigated in two ways: a double click on a subnetwork can open up that network (as in diving into a subdirectory in Windows Explorer), but an alternative is to unfold the subnetwork in-place into the parent network. The first alternative allows to work with very large designs without losing the overview, while the second alternative still enables the user to view multiple levels of a fine-grained smaller design at once.

Grouping of node connections – Upon drawing the network graph, not all connections between nodes are assigned their own graph arrows. Instead, arrows are re-used when multiple connections occur between computational nodes. This is a big space saver, as the dot program will greatly increase the graph's size as the number of connections and labels increases.

Mapping of simulation results to node size and color – By choosing a fitting mapping strategy for adjusting node's colors and sizes according to their simulation results, performance bottleneck areas can be visually identified. The internal assignment of size and color is chosen such, that *badly* performing nodes will stick out (larger, more colorful). Combined with the in-place expansion of subnetworks, this can prove to be a powerful feature.

Caching of binary event / java object translation – When drawing an event trace, a large number of events from the binary file has to be examined. Internally, an object cache is used to reduce the number of times that a java event object has to be created from its binary representation. That results in a speed increase in drawing event traces.

Time compression – The enormous time span that a typical event trace can yield, is not easily displayed in a limited amount of horizontal space. CASTviz approaches this dilemma with time compression, in which the time scale will dynamically zoom in and out according to the amount of *difference* in event types that exists.

That choice has proved to be an elegant solution. Large areas of uninteresting calculations are shrunk automatically, so that attention is drawn to the communication. A drawback is that idle time is also compressed, but by adding more nodes to the event trace, it can expand again, since there will be other nodes processing while a node is idle.

5 Applicability of CASTviz for design exploration

The concurrency measures calculated by CAST can be graphed by CASTviz in a number of ways. In this chapter, methods are proposed for using these graphing means to assist a system developer in performing design exploration steps, in order to maximize certain concurrency measures. The design exploration steps from chapter 6 of [1] are followed; part of these methods have already been applied in [5].

An engineer who is to work on an existing design, can use the hierarchical graphing module to browse through the design, getting a first look on its inner workings. Localized event traces can assist in this investigation.

The application's bar chart module was specifically designed to help keep track of the decisions made within design exploration. Performance and concurrency changes which occur in successive design versions can be displayed and followed by placing the same nodes from several design versions into a single bar chart, adjusting scaling options if necessary.

The design exploration method's various phases will be investigated in separate paragraphs.

5.1 Task splitting

The first optimization transformation consists of increasing the task parallelism for the slowest nodes in the computational network. The slowest performing nodes are split up into several new nodes, forming a computational path. The best candidate node for task splitting is the node with the lowest restart measure, because that node is limiting the throughput of the complete system. Splitting up the node will result in new nodes, each having typically a lower restart measure than the original node; this will result in a decreasing restart measure for the computational network.

Using CASTviz, pointing out the compute node with the lowest restart measure is straightforward by using the node analysis module. Selecting the restart measure, the automatic sorting will put the best candidate node for task splitting on top. For larger networks, having the *network graph mapping* function from paragraph 4.2 adjust node's display sizes or colors based on their restart measure, can assist in visually finding the bottleneck. The node analysis should then be used for a detailed analysis.

In addition, when determining the splitting strategy for specific nodes, an event trace can be created, listing the node and connected nodes. By examining idle time of the node with a high restart, it can be determined what the exact communication actions are that limit the restart measure.

5.2 Data splitting

The data tokens which a compute node processes, need not all be processed sequentially. It is often the case that no dependencies exist between a set of data tokens handled by a node.

Data parallelism can be introduced for such nodes, enabling parallel processing of the mentioned data tokens. This is achieved by replacing the original node with a *split* node, which distributes the data to a number of new parallel compute nodes, and a *join* node which receives the calculated data and synchronizes on it. Consider figure 5.1 for an example of a node transformation performed during the data splitting step, replacing a node n by a splitting node s , new processing nodes n_1 and n_2 , and a join node j .

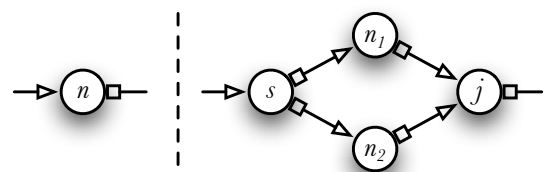


Figure 5.1: Node transformation due to data splitting

The structure measure determines the amount of data parallelism in a computational network, by examining the average number computational paths flowing through a compute node. But since the structure measure is only defined for compute networks (not for single nodes), pinpointing a candidate node for introducing data parallelism can not be done by looking at the structure measure alone. Knowledge and examination of the source code of nodes in a compute network with a low structure measure, is still required. An event trace might help reveal long chunks of internal events inside candidate compute nodes; these could be indications of possible locations for introducing data parallelism, if investigation of the source code indicates limited dependencies within the internal event series.

5.3 Communication granularity

Compute nodes which communicate heavily, perform a lot of read and write actions to and from surrounding nodes. Every commutation event introduces a delay to the node however, limiting the time spent on actual computation. The computation load measure captures this balance between communication and computation time for a compute node.

The communication granularity optimization step seeks to increase nodes' computation load by grouping low-grained communication into single read or write actions.

Similarly to the task splitting step, CASTviz will happily assist in finding nodes with a low computation load. Combining the approaches of using the network graph mapping and node analysis functions, a number of candidate nodes for optimizing communication granularity can be selected.

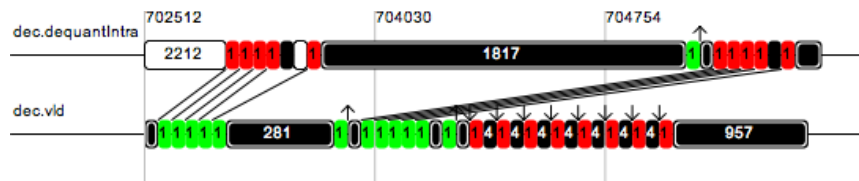


Figure 5.2: Typical event trace for low communication granularity

Event traces of these nodes should be examined. By looking for many communication actions performed in rapid succession, the code fragments responsible for low-granularity communication can be found. Figure 5.2 displays an example of such low-grained communication actions. Pointing at any communication event with the mouse will display the relevant code fragment.

5.4 Data merging

The data merging step tries to evenly distribute the amount of computing work amongst the computational paths present in a compute network. Computational paths which suffer from a low execution load can be evened out by reducing the number of compute nodes, joining together branched constructions similar to the ones introduced in the data splitting part. Correctly applying these mergers will raise the execution load of involved compute nodes.

When applying a data merging transformation, candidate branches can be selected by setting network graph mapping to vary node size with execution load. In a network with many loads or levels (onto which typically the data merging step will be applied), branches with consistently low execution loads can be visually identified, and examined as merger candidates.

5.5 Task merging

As a final step in the design exploration method, an effort is made to maximize the execution load of a compute network. Neighboring nodes which have a total summed execution load of marginally less than 1, are candidates for a merge operation. By combining the nodes, the merged node will have an execution load of almost 1, removing as well the need for the superfluous communication channel between the original nodes. Applying a series of these operations will gradually increase the execution load of the compute network.

The visualization tool does currently not calculate summations of measures for selected nodes, which by the description from the previous subparagraph, would assist in finding node pairing candidates. A more iterative approach will be required, browsing through the nodes and hierarchic levels in a compute network, looking for structures which matching execution loads. Since the task merging step can be viewed as the inverse operation of the task splitting step, structures resembling processor pipelines (nodes communicating in succession) should be investigated first.

6 Conclusion

A number of final remarks about the accomplished work in the project, are given in this chapter.

6.1 Summary of conclusions

Having applied both CAST and its counterpart CASTviz in a number of actual use case scenarios, the *design exploration method* mentioned in [1] has proved valuable in increasing the concurrency measures for a compute network under investigation. Specifically, a case study of an H.263 decoder has been performed using this workflow[5].

The visualization options in CASTviz serve a two-fold source of requirements: the direct statistical results of a computational analysis are displayed in a variety of ways, while also specific modules assist in following the design exploration method.

The CAST statistics are available in numeric form, directly in the tree view, and by comparison in the node analysis module. The displayed graph can be adjusted to highlight a concurrency measure by adjusting its node size, node color saturation or node color brightness accordingly. Various design iterations can be compared by placing relevant nodes into a bar chart.

The numerical analysis tools can be applied to assist in various steps of the design exploration method. In addition, the event trace module can provide detailed insight into inner workings of nodes, for explaining a calculated concurrency measure, or for investigating the actual line of code of a compute node causing a pin-pointed performance bottleneck.

By creating the application in a modular way, based on an extensible data model, additional functionality can be incorporated in a robust manner. A number of ideas for enhancements are mentioned in the next paragraph.

6.2 Future tasks

As the use of the visualization tool increased, a number of enhancements came to mind that will broaden the applicability of the software, or assist better in following the design exploration method. A list of possible future enhancements will now be given.

- The spatial placing of nodes and the routing of drawing connections between the nodes, is being done by the external 'dot' program right now. This limits interactivity with the graph, and limits spatial control over node placement. By creating a custom routing algorithm and performing the routing inside CASTviz, a more effective graph browsing method might be developed. A hyperbolic zoom function like in [6] might be applied to view local graph detail without losing display of the broader network context.
- The iterative work flow of a design engineer consists right now of three separate environments: the ordinary compile/build / debug environment, the running and interpretation of the CAST simulation tool, and applying the CASTviz visualization on a network. Since these steps are typically successive in a work flow, an integrated development and debugging environment can be envisioned which combines the three functions. No IDE should have to be created from scratch; an existing environment might be extended to accommodate this functionality.
- The time compression algorithm implemented in the event trace module currently has unadjustable compression decisions; time is compressed as far as to being able to draw all separate events. The result is that although all event type changes (idle / internal / communication) are very visible, the time axis is very non-linear and interpretation of event traces can become misleading if the discontinuous scale is ignored.

Less distorted diagrams might be created if some inertia is added to the erratic scale changes, along with an exact display of the current scale along the x-axis of an event trace. This will make it easier to follow the flow of time.

- When browsing an event trace, the relevant code fragment responsible for an event will be displayed when pointing at an event. Of even more interest might be displaying the complete stack trace at the time of event generation. While recording this information for the complete duration of a CAST simulation would generate way too much data, the previously proposed integration of debugger, CAST and CASTviz might be able to put a breakpoint on an exact time stamp of interest.

- While discussing the design exploration method, the general approach of CASTviz' display of CAST simulation results proved to help at some exploration steps, but not all. Specific analysis additions that can assist in design exploration, include:
 - *Visualize the computational paths* – By explicitly mentioning these data flow markers, structure-related design exploration steps like data splitting can easier point out nodes which have many computational paths in common. In addition, computational path visualization will prove to be useful as a general analysis tool of networks.
 - *Automatically search for low-granularity communication* – Two nodes which are communicating a lot of small communication tokens in rapid succession should be effortless to track down. The design engineer would then, assisted by event traces, typically change the source code iteration to use larger blocks at a time.
 - *Search for neighboring compute nodes with combined execution loads near 1* – To assist the task merging process of paragraph 5.5, a list could be created of neighboring compute nodes which are candidates to be merged. The graph display could be annotated, marking groups of nodes having a matching combined execution load.

Extending CASTviz with the mentioned features, within the scope of future internships or master's projects, will further broaden the developing field of assisted and automated design exploration. The development method and iteration steps proposed in [1] and further applied in [5], do seem applicable for automated assistance. With more performed case studies and additional CASTviz functionality, the tools can sport an intelligence that allows a design engineer to focus on optimal algorithmic development instead of coping with concurrency constraints or platform-specific optimizations.

Appendix A: Support libraries and programs

Along with the runtime libraries provided by Java, a number of additional libraries and support programs was used. All libraries and programs mentioned can be used and re-distributed free of charge.

A.1 Scrollable desktop

To allow for a familiar interface for working with a multiple document interface (MDI), a library from Tom Tessier [7] was used to manage a scrollable document area, on which multiple Swing windows are displayed.

The library provides a scrollable desktop area, a task bar for window navigation, and a familiar Window menu for window resizing, ordering and closing.

A.2 Logging for java

The logging for java, or in short log4j [8] library, provides a flexible framework for working with log (trace) messages. All logging sources are organized in a tree, and on every crossing in the tree log messages can be filtered on priority.

The system allows for very fine-grained enabling and disabling of specific log messages.

A.3 Flexible XML object retriever

Java comes with a powerful XML parsing engine called sax (Simple API for XML). It is very flexible and fast, but not usable for directly building complex tree-like structures like a computational network. During the trainee project, a simple XML retrieval was written for the visualization software, but in a modular way, so it might be used in other programs.

This paragraph will explain the class structure and usage of the XML object retriever library.

A.3.1 Class structure

The class diagram of the XML object retriever library is shown in figure A.1. A specific `XMLObjectRetriever` implementation is responsible for reading specific XML tags, and for providing a new set of `XMLObjectRetriever` implementations that is valid for the body of an XML tag.

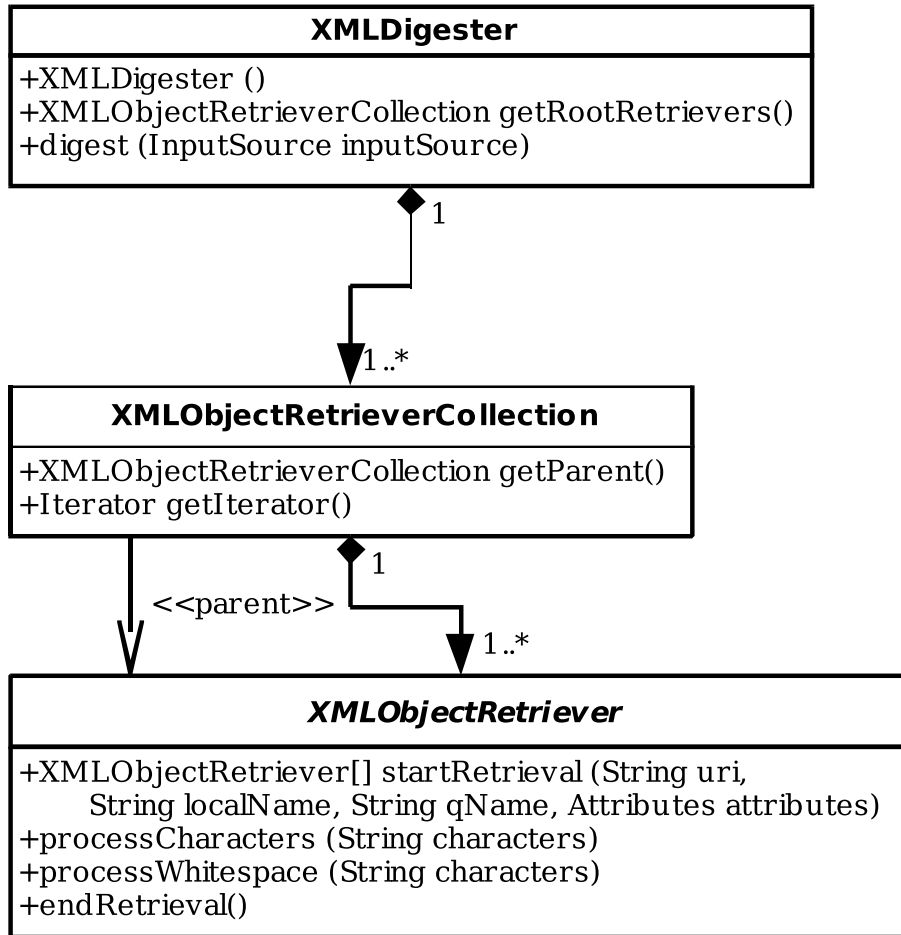


Figure A.1: XML Object retriever class diagram

An `XMLDigester` instance parses a sax `InputSource` and sends encountered tags to a set of active `XMLObjectRetriever` instances, contained in an `XMLObjectRetrieverCollection` object. At the start of XML parsing, a root `XMLObjectRetrieverCollection` is active, which should have been provided with `XMLObjectRetriever` instances valid for the type of XML being parsed.

When an XML tag is encountered, and a matching `XMLObjectRetriever` instance is found, the retriever's `startRetrieval()` method is called, and from the result of that method a new `XMLObjectRetrieverCollection` is built that is valid in the context of the XML tag. Tags which aren't matched to any `XMLObjectRetriever` are ignored.

A.4 'Dot' graph layout

The 'dot' program[9] is graph layout software that can layout a directed graph in a 2D plane in a visually pleasing way. The software can be fine-tuned by customizing colors and sizes to be used for nodes, and handles subgraphs by laying them out in a recursive way. Dot uses a custom text format as graph input language and generates a laid-out graph in this same language.

Bibliography

- [1] S. Stuijk, *Concurrency in Computational Networks*, ES, dpt. Electr. Eng., Eindhoven University of Technology, 2002
- [2] E.A. de Kock et al., *YAPI: Application Modeling for Signal Processing Systems*, Proc. 37th Design Autom. Conf., 2000
- [3] Object Management Group, *Universal Modeling Language*, <http://www.uml.org/>
- [4] A. Mazzeo, N. Mazzocca, U. Villiano, *Efficiency measures in heterogeneous distributed computing systems*, *Concurrency: Practice&exp.*, vol. 10-4, pp 285-313, 1998
- [5] R. Kneepkens, *H.263 Decoder: Case study on concurrency analysis*, ICS, dpt.Elekt.Eng., Eindhoven Univ.of Technology, 2003
- [6] T. Munzner, *Exploring Large Graphs in 3D Hyperbolic Space*, *IEEE Comp. Graph. & App*, vol.18no.4, pp. 18-23, 1998
- [7] Tessier, T., *Article and Book Excerpts*, <http://www.tomtessier.com/articles/>
- [8] Bogaert, M., *Log4j - Introduction*, <http://jakarta.apache.org/log4j/docs/>
- [9] AT&T, *Graphviz - Graph Visualization Software*, <http://www.graphviz.org/>