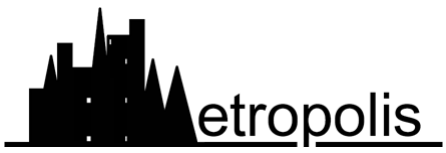
Simple Case Study in Metropolis

Haibo Zeng
Vishal Shah
Douglas Densmore
Abhijit Davare

September 14, 2004

Memorandum UCB/ERL M04/37



Contents

Abstract

The case study documented in this tutorial exercises the capabilities of the Metropolis Design Environment with an industrial-sized design. As a typical design example in Metropolis, this case study consists of a functional network, an architectural network, and a mapping network. The functional network models a simple application where data is obtained from two independent sources, manipulated in some way. This document will describe each of these three major components.

One

Introduction

The case study documented in this tutorial exercises the capabilities of the Metropolis Design Environment with an industrial-sized design. As in the platform-based design methodology [5] which is one of the basic principles espoused in Metropolis, each step of the design flow is characterized by a function, an architecture and the mapping of the former onto the latter (see Figure 1.1). The design process [9] typically starts with a *functional network* which is a denotational description of the function to be implemented, plus a set of constraints the implementation has to satisfy. The functional network specifies the application space as in Figure 1.1. An *architectural network* is also modeled as an interconnection of library components resulting in a structure that can implement a set of functionalities. The architectural network specifies the architectural space as in Figure 1.1. Then the action of *mapping* a function onto an architecture generates a new function described at a lower level of abstraction. As a typical design example in Metropolis, this case study consists of a functional network, an architectural network, and a mapping network. The following chapters will describe each of these three major components.

1.1 Audience

This document is intended to showcase a concrete example of a multimedia design within the Metropolis environment. The reader is expected to have a basic familiarity with the concepts of Platform-based

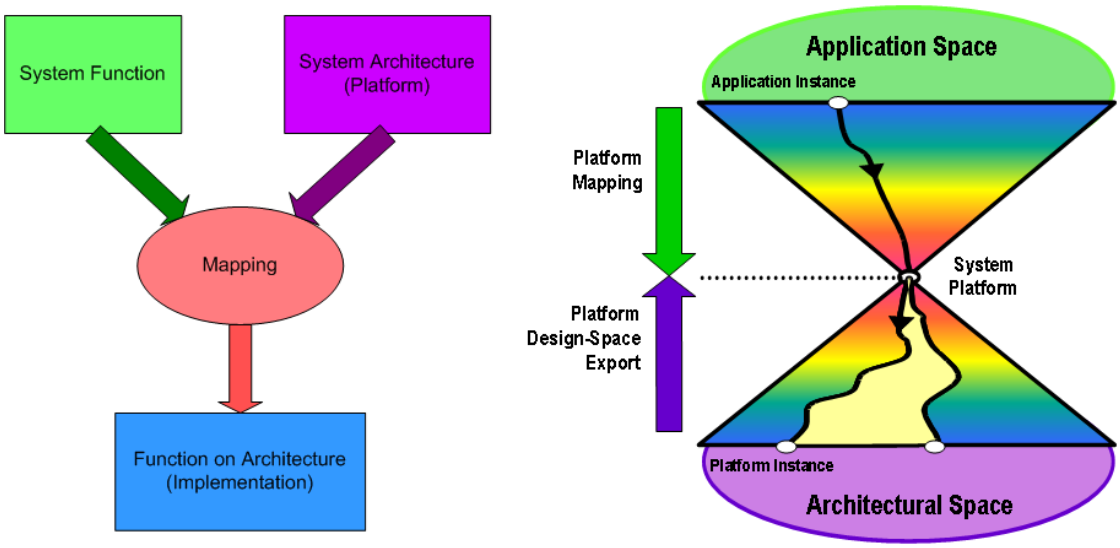


Figure 1.1: Design Process in Platform-based Methodology

design [5], the Metropolis metamodel specification language [1, 7, 11], and the Metropolis infrastructure [1, 11].

Two

Functional model

The functional network for this case study is a simplistic application which consists of two source processes, a join process, and a sink process. This application is modeled in Metropolis as a process network consisting of a number of processes and channels. The channels are modeled at various levels of abstraction, which will be described later in this chapter.

2.1 Overview

The simple application models an application where data is obtained from two independent sources. These sources write their data to independent channels. A separate process then reads a data item from each channel, possibly manipulates it in some way, and then outputs the data to another channel. Finally, a sink process reads the items from this last channel. According to user-specified parameters, the size of the items, and the number of items to be processed can be controlled. An block diagram of this funtional network is shown in Figure 2.1.

2.2 Computation vs. Communication

One of the major concepts espoused in Platform-based design is the orthogonalization of concerns. For the functional model, we focus on the orthogonalization between computation and communication. The com-

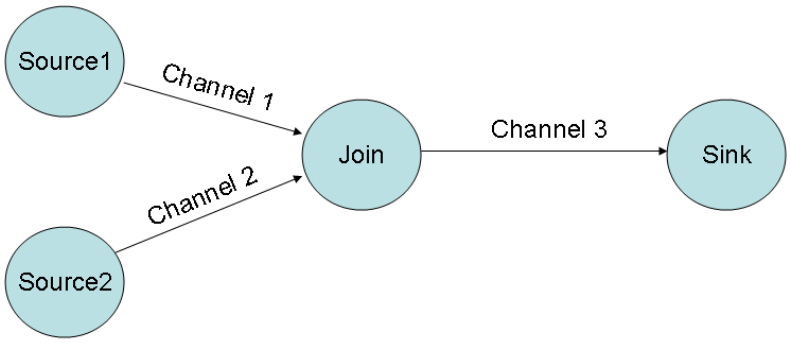


Figure 2.1:

putation portion of the application is concentrated in processes while the communication portion resides in channels. Depending on the level of abstraction, the channels are modeled using media or netlists of media.

Each process in the network represents a separate thread in the application. The processes execute concurrently. Interaction with other processes (e.g. communication, handshaking, blocking) only takes place by utilizing the services provided by the communication channels.

For most designs, the computational portion (process network) is the differentiator. Inter-process communication schemes are usually standardized across a wide class of applications. With this idea in mind, Metropolis provides users with a library of frequently-used communication schemes (channel implementations) to simplify the design process. The application in this case study makes use of two such communication schemes from the library.

2.3 Abstraction Layers

Modeling an application at different layers of abstraction allows the designer to focus on different aspects of funtional modeling as the design process progresses. At the beginning, the designer may be interested in debugging the computational portion of the design, and may not want to worry about possible deadlocks caused by the communication scheme. Later on, as the design matures, the primary goal may shift to more closely modeling the memory requirements of the functional design, with an eye towards future implementation on an architectual platform.

Following this line of thinking, the application in this case study is modeled at two major levels of abstraction: *yapi* and *TTL*. The *yapi* layer models communication with an unbounded point-to-point FIFO. The *TTL* layer also model point-to-point communication, but with bounded resources.

yapi layer

The yapi layer models point-to-point FIFO communication with unbounded resources [6]. Following the Kahn Process Networks model of computation, yapi channels have non-blocking write and blocking read semantics. Since unbounded communication resources cannot introduce deadlock in an otherwise correct functional specification, this communication layer is ideal for initial debugging of a functional model. However, the assumption of unbounded resources makes this communication layer unsuitable for mapping onto architectures with finite resources.

The interface provided by a yapi channel to the processes is very simple. Two major functions are provided: *write* and *read*. The arguments to these functions are the data arrays and the number of items to be written or read respectively. Since this communication scheme (indeed, the subsequent scheme as well) makes use of templates, the same implementation can be reused for different data types. The *write* function is nonblocking, due to the assumption of infinite resources, while the *read* function blocks until at least one data unit becomes available.

TTL layer

The TTL layer models point-to-point FIFO communication with bounded resources [3, 4]. Since the resources are bounded, TTL channels have blocking write and blocking read semantics. The user can specify the storage capacity of the TTL channel as a parameter. As long as the size of the channel is at least 1, the TTL protocol guarantees that resource limitations themselves will not cause deadlock. However, the smaller the size of the TTL channel, the more context switches that will be required to transfer the entire amount of data. This represents an interesting tradeoff between storage space and computation.

Three

Architectural model

This chapter describes the architecture networks as in Figure 1.1. These architectures are designed for the application described in Chapter 2. Also, they can be examples of how an architecture can be described using the Metropolis metamodel language.

3.1 Overview

In Metropolis an architecture is some interconnection of computational and communication resources characterized by performances and costs. It is important associate costs and performances to each component and a way of estimating costs and performances. This is very important in order to be able to explore the implementation space and compare solutions to decide which one is the best. For this simple case study, we define a simple architecture composed of a CPU/RTOS component, a bus and a memory, and a refined architecture which refines some components in the simple architecture such as the CPU/RTOS into a more detailed netlist.

3.2 Architectural Execution Semantics

One of the major concepts espoused in Platform-based design is the orthogonalization of concerns. This can also be a solution to the reuse prob-

lem. There are several concerns in embedded system design that can be orthogonalized, such as behavoir versus architecture, computation versus communication. Even within a model of architecture there are two aspects that could be represented separately, capability, i.e. the set of behaviors the architecture can implement versus the cost it bears when it implements a given behavior. For example, in modeling a CPU in terms of the instructions it supports, one would capture the behavior of each instruction such as addition or data move, as well as the cost of the instruction such as the number of required clock cycles or latency in the local time defined for the CPU.

To facilitate this orthognolization, in Metropolis methodology, a typical architecture model consists of the *scheduled network* that models all the capabilities, and the *scheduling network* that models performance and coordination between components of the architecture. The architecture network then executes in two phases, the request phase and the resolution phase. This two-phase execution semantic allows us to use quantity managers not only for annotating behavior with quantity but also for modeling scheduling policies for shared resources. Quantity managers, as a part of their syntactic requirements [11], implement two methods, $request(...)$ and $resolve(...)$, corresponding to the two phases. The request method allows a designer to describe the queuing model he/she wants to use, and the resolve method can model the scheduling policy.

In the request phase, architectural components instantiated in the scheduled network enqueue the events and the quantity amount with which these events should be annotated to the quantity managers. After all the requests from the scheduled network have been made, the resolution phase starts, and in this phase the scheduling network will iteratively call the $resolve(...)$ method of each quantity manager until all of them reach a stable decision on which tasks to let run.

In Section 3.3, an example is shown to explain the execution of architectural network.

3.3 A Simple Architecture

Netlist

The structure of the simple architecture is provided in Figure 3.1. In this abstract architecture model, there are three types of media: a CPU/RTOS, a Bus, and a Memory.

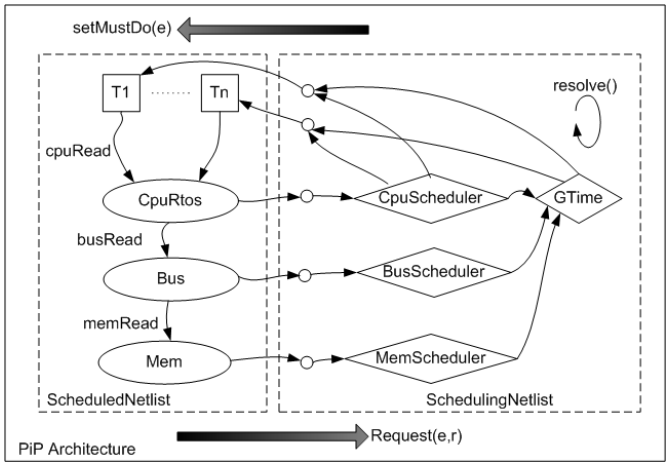


Figure 3.1: Structure of the simple architecture

The tasks in the architectural model are also called *mapping processes*, which serve as the interface between the architectural components and the functional model during the mapping stage. The tasks nondeterministically model all the possible programs that could be executed on the architecture. The software tasks are connected to the CPU/RTOS and use the services that those components offer for computation or communication. The CPU/RTOS is connected to the Bus. This component is a multi-master multi-slave communication device that allows the CPU/RTOS to communicate with the Memory. The Memory is a slave device connected to the Bus. The CPU/RTOS component is shared among several tasks. Each of these could require a service. When more than one request is issued to the CPU/RTOS, arbitration is needed. We can think of the CPU as a device that must be distributed among the tasks. Each time there is a request from a task, the request is passed to the CpuScheduler quantity manager which decides the task that will be the owner of

the CPU/RTOS. The same procedure occurs when more than one master asks for ownership of the Bus or Memory. In this case, the BusScheduler or MemScheduler quantity manager will decide the winner. There is another quantity manager called GTime which annotates instances of events with the Global Time physical quantity.

As mentioned in Section 3.2, the architecture is composed of two netlists: a scheduling netlist and a scheduled netlist. Each process and medium in the scheduled netlist has a statemedium associated with it. These statemedium belong to the scheduling netlist. The scheduled netlist also contains all of the quantity managers. The CPU/RTOS or the bus media can issue requests to the quantity managers by calling the *request(...)* method of corresponding statemedium. The user defines the statemedium and the implementation of the request method. The statemedium request method will call the request method implemented in the quantity managers. This method will store all the requests in order to make a decision on which one will be satisfied. The quantity managers resolve all the conflicts in the requests and decide the next event vector that will make the state transition in the scheduled netlist. In order to specify the event vector, the quantity managers can call the methods *setMustDo(e)* or *setMustNotDo(e)* implemented by the statemedium which connects to the software tasks (where e is the event). Basically *setMustDo(e)* says that for this quantity manager, if there are some event that can be granted in this execution phase, it can only be e (but whether e can be granted is dependent on the decision of other quantity managers); *setMustNotDo(e)* says that e can't be granted in this execution phase.

Components

In this section all the components in the architectural model are explained in more details.

Data Types

In order to pass requests from scheduled netlist to scheduling netlist, we define a request class *SchedReqClass* which contains the useful information about the request. Based on it, the corresponding quantity managers will make their decisions such as arbitration or quantity annotation.

```
public class SchedReqClass extends RequestClass {
```

```
  private event  _requestEvent;
  private event  _referenceEvent;
  private int    _serviceID;
  private int    _nService;
  private int    _masterID;
  private int    _slaveID;
  private double _time;
  private int    _prio;
}
```

$\_requestEvent$ is the event this request is made for. $\_referenceEvent$ is the event which can be a reference event to the $\_requestEvent$. $\_serviceID$ specifies the Id representing the service associated with this request, for example, CPU_REQUEST which is a computation service provided by the CPU component. $\_nService$ is the number of atomic services to be satisfied. $\_masterID$ is the id of the master which calls this service interface function. $\_slaveID$ is the id of the slave whose service will be called to accomplish this request. $\_time$ is the time stamp to be annotated with the $\_requestEvent$. $\_prio$ specifies the priority associated with this request, which is essential in some priority-based scheduling algorithm.

Software Tasks

Software tasks in the architectural model are also called *mapping processes*, which serve as the interface between the architectural components and the functional model during the mapping stage. The tasks provide to the functional model all the programs that could be executed on the architecture. To complete all the capabilities needed by functional model, the software tasks are connected to other components such as media and use the services those components offer for computation or communication.

In the architectural models, the software tasks offer two types of services: a generic computation service *execute()* which is an execution of a specific functionality associated with a parameter on how complex it is, and two communication services – generic read and write.

```
public process SwTask {
  public void execute(funId, comp)
  public void read (base, offset, objSize, numObj)
```

```
  public void write(base, offset, objSize, numObj)
}
```

CPU/RTOS

The medium CPU/RTOS implements the services used by the software tasks. One basic service is the request of the cpu ownership for a certain number of clock cycles. In this simple model this service can be used to model computation. We need also some communication services like read and write. Here is the definition of the services offered by the CPU/RTOS to the software tasks:

```
public interface SwTaskService extends Port {
  eval   void request(n);
  eval   void read (target, addr, n);
  update void write(target, addr, n);
  eval   void readLong (target, addr, n, data);
  update void writeLong(target, addr, n, data);
  eval   void readProtect (target, addr, n);
  update void writeProtect(target, addr, n);
  eval   void readLongProtect (target, addr, n, data);
  update void writeLongProtect(target, addr, n, data);
}
```

The CPU/RTOS medium definition is as follow:

```
public medium Cpu implements SwTaskService {
  port SchedReq   _portSM;
  port CpuSlave[] _portSlaves;

  //Methods implementation
  ...
}
```

The Cpu medium has two types of ports: *_portSM* is the port that will be connected to the statemedium. The cpu can make requests through this port to the corresponding quantity managers. The ports *_portSlaves* will be connected to the busses which have to implement the *CpuSlave* interface (see section 3.3).

As an example, the implementation of the request service is shown as below. *request(...)* service can be used by a task to request the CPU/RTOS ownership for a certain number of cpu cycles. The implementation is as follow:

```
public eval void request(int n) {
  {$
    beg{
      e = beg(getthread(), this.request);
      _src.setSchedReqClass(e, SERVICE_ID_CONTEXT_SWITCH, 1, -1, -1);
      _portSM.request(e, _src);
    }
  $}

  creq{@;
  {$
    beg{
      e =  beg(getthread(), this.creq);
      _src.setSchedReqClass(e, SERVICE_ID_REQUEST, n, -1, -1);
      _portSM.request(e, _src);
    }

    end{
      r =  end(getthread(), this.creq);
      _src.setSchedReqClass(r, e, SERVICE_ID_RELEASE, 1, -1, -1);
      _portSM.request(r, _src);
    }
  $}@};
}
```

At the beginning of the method, a request asking to be the owner of CPU/RTOS is issued to the CpuScheduler quantity manager through the corresponding statemedium. Then a request asking for a certain number of cpu cycles' computation is issued. At the end, the owner will try to release the CPU/RTOS medium by another request. The two events $(beg(getthread(), this.creq)$ and $(end(getthread(), this.creq)$ will be annotated with two different timestamps and the latency between them will be the number of cycles requested (if no preemption occurs). The

CpuScheduler quantity manager will take care of annotating the events. The read and write services are offered in many different flavors. There are simple read and write that take as parameters the target device and the number of bytes. There is also a protected version meaning a version where the target address is exclusively accessed by the current owner until the read or write operation ends.

Bus

Similar to CPU/RTOS, the Bus medium implements the *CpuSlave* interface which offers services its masters), and has an array of ports to connect to its slaves.

Memory

The memory is simply a slave component. It is connected to the bus meaning that it implements the *BusSlave* interface and the bus has a port connected to the memory.

Quantity Managers

The Metamodel facilitates cost annotation of architectural services and modeling scheduler policies for shared resources using special objects called *quantity managers*. The quantity manager objects provide a clean way to separate the functionality offered by an architecture from the estimate of its performance and its scheduling. Figure 3.2 shows a high level anatomy of a quantity manager. Its input can be seen as a set of annotation requests from the software tasks wanting a share of the quantity modeled by this manager. These requests are queued in using the *request(...)* method implementation and its output can be seen as either a single result, or a sequence, indicating the execution order of the requesting tasks. The algorithm for scheduling the access to the shared resource is modeled in the *resolve(...)* method of the quantity manager.

The release includes models of two schedulers, a time-slice based scheduler and an FCFS based FIFO scheduler found in *SchedulerTimeSliceBased.mmm* and *SchedulerFIFO.mmm* respectively. The scheduler models are independent from the models of cpu, bus and memory and can be used with either of them. The reader should follow the resolve method

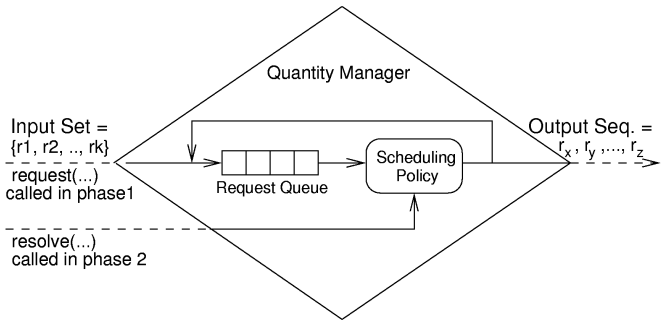


Figure 3.2: Quantity Manager Anatomy

of each of scheduler model to understand the scheduling policy it implements. The architecture models are modular enough to allow a single parameter change to switch between either of the schedulers. The following section discusses the time-slice based scheduler model.

Time-slice based scheduler model

We describe the basic ideas and the algorithm used in the time-slice scheduler model found in SchedulerTimeSliceBased.mmm. In order to achieve efficiency in simulation time, our model seeks to reduce the number of calls to the scheduling network. On these lines, a software task, in order to estimate the performance of its code, can request the annotation of a set of sequential instructions, eg. a basic block. Using methods as proposed in [8, 10, 12]. a designer can construct a model for software, in which the cycle usage of all the instructions in a basic block is aggregated, and a single request for the time annotation of this aggregated amount of cpu cycles is made to the quantity manager. We focus on modeling a time-slice task scheduler when the annotation granularity of the processes to be executed is coarser than the time-slice, as can happen in the case outlined above. The model is applicable to different kind of shared resources, such as cpus, buses, memories, etc., but in the following we will concentrate, without loss of generality, only on cpus, since this is by far the most common case.

Our goal is to avoid starting a new resolution phase each time a time-slice elapses, otherwise we would loose possible gains in simulation performance that can be achieved using coarser annotations at the process

level. When the annotation granularity is fine, on the other hand, our technique, while still applicable, does not provide significant advantages, since the bottleneck is in other parts of the system. Algorithm 1 details the resolution mechanism to be implemented in quantity managers to model time-slice based task schedulers. At the start of resolve phase each quantity manager, instantiated in the *scheduling network*, will have a set of requests from the software tasks that wish to use the resource controlled by the quantity manager. Based on the scheduling policy it models, the quantity manager will decide which task should be allowed to use the resource. *Step 1* of the algorithm selects the request ***R*** from the input set ***X***, containing the task requests.

To model time-sliced behavior when the annotation granularity can be coarser than instruction-level, our quantity manager model instead of choosing a single software task as its scheduling decision, will choose a sequence of tasks. The sequence corresponds to the alternation of execution slices of the various tasks mapped to the shared resource. The sequence stops, and the *resolve()* method terminates, when the last slice selected completely fulfills the request of a task. This task is then signaled to proceed, while all the others are kept suspended since they need more slices to finish. *Step 2* models this.

It first checks whether the resource amount requested by the selected software task exceeds the *time_slice* defined for the resource. If it does then the request is updated to reflect that the task was alloted resource amount = *time_slice*, and is put back in the request set ***X***. It then records the selected task in a sequence ***Y*** and the control goes back to step 1.

When a task is selected whose requested resource amount $<=$ *time_slice*, the task is recorded in the sequence ***Y*** and control is transferred to *Step 3*, which exits with ***Y*** as its scheduling decision. If t_1, t_2,..., t_n denote ***n*** software tasks respectively, and the value of ***Y*** is t_1, t_2, t_1, t_2, t_1 then the scheduling decision can be seen as, t_1 got the entire amount of resource it had requested for and that it can proceed, while t_2 got resource amounting to two time slices and is waiting to get more resource. In reaching this scheduling decision, t_1 and t_2 were each preempted twice.

The value of ***Y*** represents the execution order of tasks. In this case the order of execution of tasks is t_1, t_2, t_1, t_2.

Algorithm 1: Select resource owners

Input: Set X of requests from software tasks

Output: Sequence Y representing the execution order of software tasks

```
1. R = remove(X, selection_policy)
   where selection_policy = FCFS, priority based, etc.
2. if requested_amount(R) > time_slice
     reqeuested_amount(R) =
         reqeuested_amount(R) - time_slice
     insert(X, R)
     insert(Y, task_owner(R))
     go to 1
   else
     insert(Y, task_owner(R))
     go to 3
3. return Y
```

Example

Consider the HW/SW model of Figure 3.3. Here 3 tasks *A*, *B* and *C* request cpu cycles. Suppose *A* requests 10 cycles, *B* 20 and *C* 30 cycles and the time-slice for using the cpu is defined as 10 cycles. Figure 3.4(a) shows the state of request queue of cpu time quantity manager after the execution of *phase 1* . The number next to each task name indicates the number of cycles the task is waiting to be allocated.

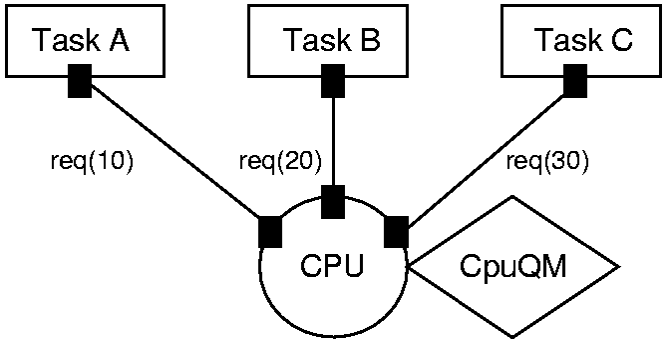


Figure 3.3: Shared Cpu

In this case, our model of scheduler, based on FCFS selection policy, will allocate first 10 and the next 10 cycles to *C* and *B* respectively, and then put them back in the queue since their entire quantity request has not been filled. Figure 3.4(b-d) show the state of the queue at the end of each time slice. At the end of 3 time slices the scheduler has with it at least one request, that of task *A* whose entire request for quantity has been satisfied. The cpu quantity manager will remove the request of task *A* from the queue and transfer the execution control to it. Figure 3.4(d) shows the status of the queue after this scheduling decision.

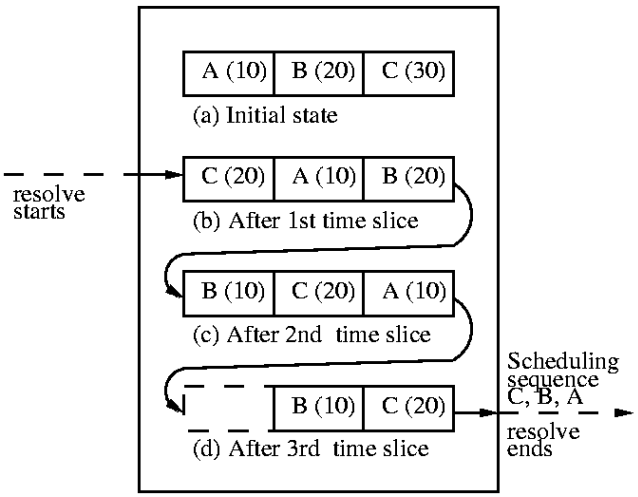


Figure 3.4: CpuQM Request Queue state

It should be noted here that in theory, the control should have transferred to tasks *C* and *B* at the end of 10 and 20 cpu cycles respectively, but the coarseness of the annotation requested by the designer did not make this a requirement. If the designer wants the control to be transferred to the software task at the expiry of each time slice, his annotation requests should within time slice defined for the resource. This allows him to introduce more details in his model at the cost of simulation efficiency. Our quantity manager can handle this case without requiring any changes in the modeling code.

Execution Example

We take Figure 3.5 as an example to explain how the architectural network executes. The numbers beside the function names identifies the sequence to call them during this execution example.

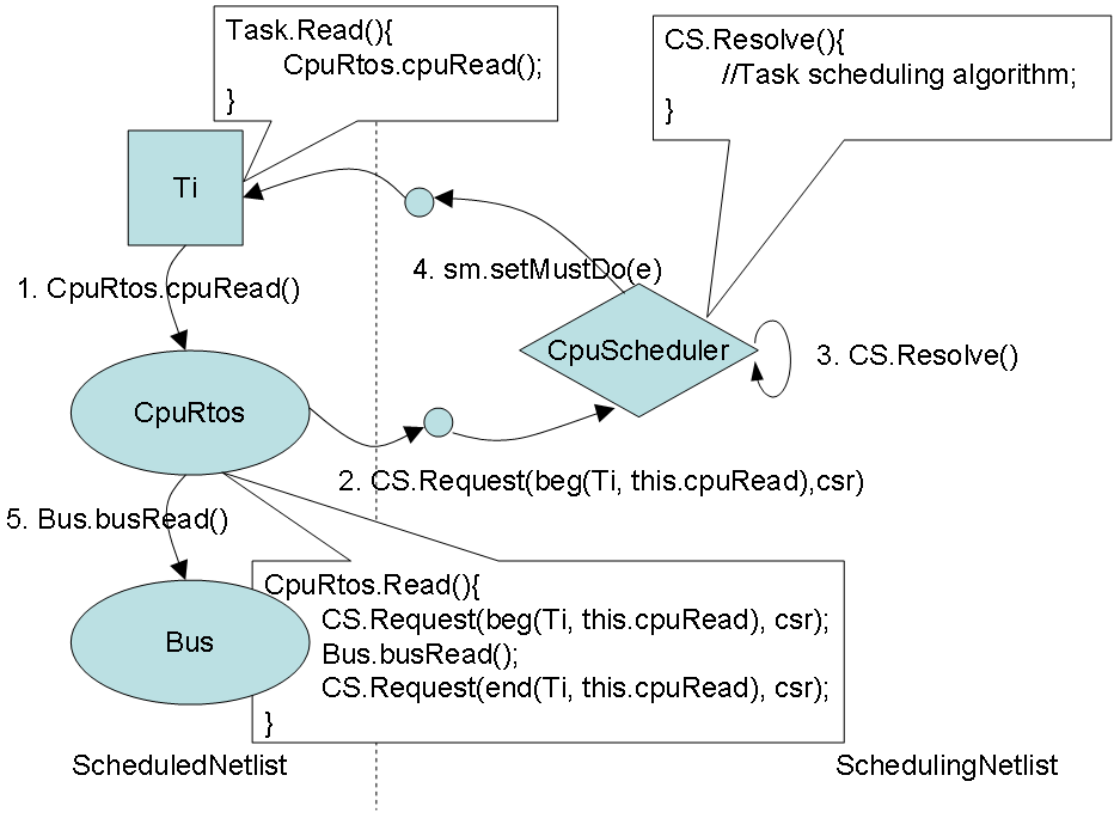


Figure 3.5: Architecture Execution Example

1. Assume at some point one of the software tasks, say T_1 calls the function *read(...)* to read some data from the memory. Within the *read(...)* method of software task, to finish this operation, it calls the *cpuRead(...)* service implemented in the CPU/RTOS medium.

2. Through the statemedium connected to CPU/RTOS, it calls the *request(...)* to make a request to the CpuScheduler quantity manager which arbitrates its access.

3. The control now passes to the scheduling netlist (assuming that all other software tasks are also waiting for some abitration or annotation). Now the scheduling netlist is executed and the *resolve(...)* methods of the quantity managers is called cyclically until the solution is stabilized.

4. The result, in our simple example, would be that T_1 will be the next owner of the CPU/RTOS. The interface function *setMustDo(...)* in the statemedium will be called to notify the scheduled network that the corresponding event is granted and T_1 can proceed.

5. T_1 now goes to the next line within the *cpuRead(...)* method; Otherwise, it will be stuck and call *request(...)* to make a request again.

Configurability

The configurability is an essential property in architecture modeling such that the designer can easily choose from a reasonable number of candidate architectures. This can leverage the automation of architecture configuration, thus facilitating the architecture exploration during mapping stage.

In this simple architecture, all the configuration can be done by instantiating an architecture and specifying the following arguments of its constructor. For a concrete example, please refer to Top.mmm in the simple architecture.

Parameterize architecture components

Each component in the architecture has some properties that characterize its performance. For example, at the abstraction level where the simple architecture is modeled, a CPU/RTOS component can be configured by some parameters as the cpu clock cycle, the OS scheduling algorithm, and the number of cycles a specific service will cost. These parameters are easily configurable as some arguments in the constructors of CPU/RTOS medium and CpuScheduler.

Specify architecture structure

Also, we need to specify how these architecture components are connected. In this simple architecture, the user can specify the connection by

some ńatural language, e.g. saying that T_0 connects to Cpu_0. The appropriate number of statemedia and quantity managers and their connections, e.g. the connection between statemedia and processes/media will generated automatically. Of course, the connection should satisfy some property, for example, if a CPU/RTOS is connected to a Bus through a port *CpuSlave*, the Bus must implement the *CpuSlave* interface.

3.4 A Refined Architecture

Once the initial architecture has been created, a natural desire is to create derivate architectures. Instead of randomly creating models, it is advantageous to do this in a disciplined manner which promotes the ability to make statements regarding the level and type of refinement performed. In addition, this approach can lend itself to formal verification techinques such as property checking. This section describes briefly one refined architecture provided for the simple case study. This is a *Vertical Refinement* model and is meant to provide an alternate, lower level of abstracion model as compared to the model described previously.

Vertical Refinement

Vertical refinement is the notion that within the model changes are made ”vertically” where these changes are additions/subtractions/divisions of media. This will consist of topological changes to existing media as well. This means that you do not swap aspects/relationships/devices between netlists but rather you move within a particular netlist. Naturally this contrasts to horizontal refinement.

Vertical refinement of an architecture can be seen as a whole spectrum of refinement with the levels being defined as to what elements are passive (media) and which are active (processes). For example you can change the number and types of processes in the scheduled netlist or you can change the number and type of media in the scheduled netlist.

What this would imply is that the most abstract would only have the clock be the only active element while the least abstract would have all processes (vice versa depending on perspective).

The primary method of vertical refinement in practice is likely to be the addition of media. This ultimately is the addition of **services**. This

is adding a level of granularity to the abstract services provided initially. Vertical refinement is most likely the most common form of refinement from a structural standpoint concerning the netlists. This is also the most straightforward of the refinement styles. This will require the following types of changes:

- Need to add/create services themselves
- Need to add requests for each services
- Need to add ”schedulers” for these services. There is a one-to-one correspondence between the services and schedulers.
- Need to introduce these into the corresponding netlists

Notice that with a vertical refinement you are moving vertically in both netlists. For example the addition of a service in the scheduled netlist requires an additional scheduler in the scheduling netlist.

Refinement Details

Figure 3.6 shows the changes introduced by the vertical architecture model.

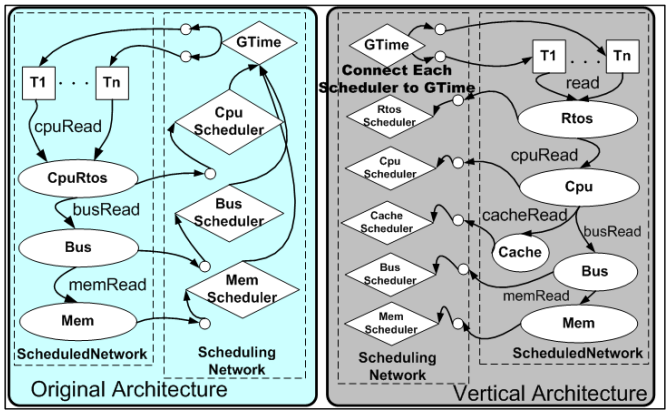


Figure 3.6: Original vs. Vertical Architecture

The vertical architecture provided adds two services:

- RTOS service. This provides an separation between the CPU and the service which schedules the tasks to the CPU. This uses the FCFS and Time Slice Based Scheduling mechanisms.
- Cache service. This provides an alternate way to service memory requests as opposed to going over the bus and to physical memory.

For more information detailing the specifics please see the document [2].

Four

Mapping

Mapping in Metropolis associates a functional model with an architectural model such that the events corresponding to services in both models are synchronized with each other. The mapped implementation inherits the quantity annotations given by the architectural model and the ordering on the service usage by processes as specified by the functional model. Also, the mapping itself may add additional information in the form of service configurations which is also inherited by the mapped implementation. This section will describe how mapping is realized in the Metropolis Metamodel for this case study.

4.1 Mapping Overview

The top-level netlist for mapping usually instantiates both the functional and architectural models with the required parameters. Events are then gathered from both models which correspond to the services that comprise the system platform. Then, synchronization constraints are added with the *synch* keyword to synchronize these events.

The design space for the system can be explored in the mapping netlist by changing the way in which the architectural and functional models are instantiated, by choosing the events to map together, and by configuring the services which these events correspond to.

The architectural model may be instantiated in different ways indicating the number of CPUs, buses and memories to be used and also the type of scheduling algorithm to be used in the CPU. The events which are to be synchronized indicate the begin and end points of different services in the system platform. By choosing the appropriate events (e.g. the beginning of a read service in a particular task and the beginning of a read service for a particular media from a particular process) to synchronize, we can indicate the assignment of functional processes to architectural tasks.

Finally, the services which are mapped together may have parameters that we can configure. For instance, a read service may have parameters indicating the number of items to be read, the size of each item, and the base address to start reading these items from. The functional process may fix the first two parameters when it uses the service, but may have no concept of an address space or base address. In this case, when we map the functional service to an architectural service, we need to specify the base address to the architectural service explicitly as part of the synchronization constraint. This indicates a mapping choice, perhaps the assignment of a storage area in the functional model to a particular physical memory in the architecture.

4.2 Mapping Details

In the mapping netlist for this case study, we first instantiate the functional model. From the functional model, we obtain the references to the processes within it. We then instantiate the architectural model with the number of tasks equal to the number of functional processes. We then also obtain the references to the architectural tasks.

At this point, we have two arrays of equal size with references to the functional processes and the architectural tasks. We are now ready to create a one-to-one association between these two sets and map each functional process onto a single architectural task. This will be accomplished by associating the events of the read and write services together.

However, we first need to gather the events corresponding to these services from the process and task arrays that we currently have. Remember that in the functional model, these read and write services are located in the storage media inside the TTL netlists which interconnect

processes. By using metamodel netlist accessor functions, we can traverse the functional netlist, starting from processes, to obtain references to these storage media, and ultimately the events which we need. There are two events we need for every service in the functional model (a read or a write by a particular process in a particular media) corresponding to the begin and the end of the service function.

After we have all the read and write events from all the media for a particular functional process and the read and write events from the corresponding architectural task, we are ready to synchronize them together. We accomplish this by using two sets of one-way synch constraints.

The first set of constraints specifies that each one of the functional events implies the corresponding event in the architectural model. If these events refer to the beginning of a service, we also include a variable equality section in the one-way synch constraint that indicates the assignment of variables. Some of these variables (number of items, data offset) are simply assigned from the functional model to the architectural model. Others, such as the base address, are specified as constants in the synchronization constraint. If the events refer to the end of the service, then no variable equality portion is required.

The second set of constraints states that the architectural event implies any non-zero subset of the functional events (OR statement). Since the services have been configured with the previous set of constraints, no variable equality is needed in this set of constraints.

Metropolis Acknowledgement

This work was supported in part by the following corporations:

- Cadence
- General Motors
- Intel
- Semiconductor Research Corporation (SRC)
- Sony
- STMicroelectronics
- and the following research projects:
  - NSF Award Number CCR-0225610 and the Center for Hybrid and Embedded Systems (CHESS, http://chess.eecs.berkeley.edu)
  - The MARCO/DARPA Gigascale Systems Research Center (GSRC, http://www.gigascale.org)

The Metropolis project would also like to acknowledge the research contributions by:

- The Project for Advanced Research of Architecture and Design of Electronic Systems (PARADES, http://www.parades.rm.cnr.it/) (in particular Alberto Ferrari)
- Politecnico di Torino
- Carnegie Mellon University
- University of California, Los Angeles

- University of California, Riverside
- Politecnico di Milano
- University. of Rome
- La Sapienza
- University of L'Aquila
- University of Ancona
- Scuola di Sant'Anna and University of Pisa

Metropolis contains the following software that has additional copyrights. See the README.txt files in each directory for details

examples/yapi_cpus/arm/arm_sim arm_sim is an ARM processor simulator that was originally released under the GNU Public License. The ARM Simulator is only necessary if you would like to create your own trace files. Most users need not build the ARM Simulator.

src/com/JLex JLex has a copyright that is similar to the Metropolis copyright.

src/metropolis/metamodel Portions of the Java code were derived from sources developed under the auspices of the Titanium project, under funding from the DARPA, DoE, and Army Research Office. The Java code was further developed as part of the Ptolemy project. The Java code is released under Metropolis copyright.

src/metropolis/metamodel/frontend/Lexer Portions of JLexer are: "Copyright (C) 1995, 1997 by Paul N. Hilfinger. All rights reserved. Portions of this code were derived from sources developed under the auspices of the Titanium project, under funding from the DARPA, DoE, and Army Research Office."

src/metropolis/metamodel/frontend/parser/ptbyacc ptbyacc is in the public domain.

Bibliography

[1] Felice Balarin, Luciano Lavagno, and et al. Concurrent execution semantics and sequential simulation algorithms for the metropolis metamodel. *Proc. 10th Int'l Symp. Hardware/Software Codesign*, pages 13–18, 2002.

[2] Doug Densmore. Metropolis architecture refinement styles and methodology. In *Technical Memorandum UCB/ERL M04/36, University of California, Berkeley, CA 94720*, September 14, 2004.

[3] Om Prakash Gangwal, Andre Nieuwland, and Paul Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In *ISSS*, pages 1–6, 2001.

[4] Om Prakash Gangwal, Andre Nieuwland, and Paul Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In *ISSS*, pages 1–6, 2001.

[5] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[6] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. Yapi: application modeling for signal processing systems. *Proceedings of the 37th Design Automation Conference*, 2000.

[7] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, Dec. 1998.

[8] S. Malik, M. Martonosi, and Y.T.S. Li. Static timing analysis of embedded software. In *Proceedings of the Design Automation Conference*, pages 147–152, June 1997.

[9] Alessandro Pinto. Metropolis design guidelines. In *Technical Memorandum UCB/ERL M04/40, University of California, Berkeley, CA 94720*, September 14, 2004.

[10] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proceedings of the Design Automation Conference*, pages 605–610, June 1996.

[11] The Metropolis Project Team. The metropolis meta model version 0.4. In *Technical Memorandum UCB/ERL M04/38, University of California, Berkeley, CA 94720*, September 14, 2004.

[12] V. Zivojnovic and H. Meyr. Compiled hw/sw co-simulation. In *Proceedings of the Design Automation Conference*, pages 690–695, June 1996.